VNIVERSIDAD NACIONAL AVTNOMA DE MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"Verificación simbólica de redes genéticas mediante una lógica temporal híbrida"

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS
(COMPUTACIÓN)

P R E S E N T A:

José Julián Argil Torres

DIRECTOR DE TESIS: Dr. David Arturo Rosenblueth Laguette

México, D.F. **2009.**

CONTENIDO

CAPÍTULO 1

Introducción

Este trabajo de tesis tiene como objetivo desarrollar un verificador de modelos simbólico que permita la verificación de propiedades de redes genéticas representadas en forma de redes booleanas. La técnica de verificación de modelos ha demostrado ser un enfoque ideal para validar propiedades de sistemas grandes; en específico ha demostrado su potencial en el desarrollo de sistemas digitales, a tal grado que hoy en día es inconcebible que empresas como Intel o AMD diseñen nuevos circuitos integrados sin ayuda de la verificación de modelos.

Para desarrollar el verificador de modelos es necesario que entendamos algunos conceptos de lógica, en específico de lógica temporal. También es necesario estudiar algoritmos que pueden aplicarse para usar cierta lógica temporal sobre representaciones de sistemas reales. Es asimismo importante comprender los motivos que impulsaron el desarrollo de nuestro verificador en redes genéticas y finalmente entender la implementación del verificador en un lenguaje de programación.

En el capítulo 2, titulado *Verificación de Modelos*, discutimos distintos enfoques de lógicas temporales. El capítulo 2 comienza con un primer enfoque hacia las lógicas temporales sin adentrar en los detalles, simplemente con el objetivo de dar paso a las dos lógicas que nos interesa entender: *LTL* y *CTL*. La primera lógica que discutimos con detalle lleva el nombre de *Linear-time Temporal Logic (LTL)*. En dicha sección presentamos al lector la sintaxis de *LTL* en forma Backus-Naur, y posteriormente mostramos la semántica de dicha lógica mediante la noción de satisfacción. La segunda lógica lleva el nombre de *Computation-Tree Logic (CTL)*. Para dicha lógica una vez más presentamos la sintaxis en forma Backus-Naur y la semántica, pero esta vez apoyándonos en la semántica definida para *LTL*. A lo largo de la explicación de ambas lógicas (*LTL* y

CTL) introducimos algunas definiciones adicionales que sirven para entender completamente la discusión. También mostramos algunos ejemplos con el objetivo de aclarar las ideas y de ilustrar en términos prácticos algunas aplicaciones de cada lógica.

Como parte del capítulo 2 hacemos una breve pero importante discusión sobre las ventajas de utilizar ya sea *LTL* o *CTL*. Esta parte es medular pues aclaramos la razón por la cual el verificador se basó en *CTL* en lugar de *LTL*. Más tarde, también en el capítulo 2, introducimos la idea de "hibridación" de *CTL*: partimos de una presentación histórica y seguimos con una explicación usando el mismo estilo con el cual presentamos las versiones originales de *LTL* y *CTL*.

La presentación del algoritmo que nos permite la automatización de la verificación de modelos está en la sección 7 del capítulo 2. En dicha sección damos por hecho que se entiende *CTL* y con eso en mente introducimos los algoritmos para automatizar la verificación con respecto a ciertos operadores primitivos; para otros operadores hacemos uso de las equivalencias presentadas durante la discusión de *CTL*.

Finalmente para concluir el capítulo 2, presentamos la estructura de datos que soporta la verificación simbólica implementada en nuestro verificador: diagramas binarios de decisión (*BDDs* por sus siglas en inglés). Para esta estructura de datos presentamos algunos algoritmos esenciales, como lo son *exists* y *apply*. Al final presentamos la forma en que serán utilizados dentro de nuestra implementación de verificador simbólico.

Para el capítulo 2 suponemos que el lector tiene conocimientos de lógica proposicional, lógica de primer orden, estructuras de datos y algoritmos. El capítulo 2 tiene como objetivo mostrar al lector una visión general del verificador de modelos desde una perspectiva más bien teórica.

El desarrollo del verificador tuvo siempre como objetivo poder verificar redes genéticas; para cumplir con este objetivo contamos con dos casos de estudio. El primer caso de estudio se basa en los resultados publicados por el grupo de Elena Alvarez-Buylla del Instituto de Ecología de la UNAM sobre la flor de la planta *Arabidopsis thaliana*. El segundo caso de estudio no cuenta con publicación aún; el objetivo en dicho caso de estudio es obtener los resultados para la raíz de la planta *A. thaliana* haciendo uso de nuestro verificador y posteriormente publicarlos. El primer caso de estudio tiene un doble interés para nosotros: por un lado queremos mostrar el uso de verificación de modelos sobre redes genéticas reales; por otro, deseamos comprobar los

resultados arrojados por nuestro verificador contra resultados públicamente validados con lo cual podemos validar nuestra implementación.

Realizamos la presentación de los casos de estudio en el capítulo 3, titulado *Redes genéticas*. Como introducción para los casos de estudio y para tener un panorama general del entorno sobre el cual verificamos, presentamos una discusión sobre redes genéticas, introducimos algunos conceptos más bien biológicos y mostramos el ciclo general de modelado y simulación que actualmente utiliza la biología.

También como parte del capítulo 3, presentamos una discusión sobre los principales formalismos matemáticos que utiliza la biología para representar redes genéticas. Los formalismos que presentamos son las gráficas dirigidas, las redes bayesianas, las redes booleanas y finalmente las redes lógicas generalizadas. Concluimos la presentación de los formalismos con una discusión sobre por qué elegir un formalismo en lugar de otro.

El capítulo 4, titulado *Verificador*, tiene como objetivo presentar el verificador en términos prácticos, haciendo hincapié en nuestra implementación a través de ejemplos extraídos directamente del código fuente de nuestro verificador de modelos. El capítulo 4 es la documentación necesaria para continuar el desarrollo del verificador en trabajos futuros. Presentamos la arquitectura del sistema y después nos adentramos en cada capa de la misma. En cada capa nos apoyamos en diagramas basados en el Lenguaje Unificado de Modelado (*UML* por sus siglas en inglés); en particular explotamos los diagramas de clase. El verificador mediante el capítulo 4 puede ser entendido como una interfaz de programación de aplicaciones (*API* por sus siglas en inglés) que permita el uso de verificación de modelos en una infinidad de aplicaciones.

Para el capítulo 4 suponemos que el lector tiene conocimientos de programación orientada a objetos y del lenguaje de programación C#; hacemos uso de la versión 2.0 de dicho lenguaje.

Para el desarrollo del verificador utilizamos la interfaz de desarrollo Visual Studio 2005 Professional, por lo cual la solución del verificador deberá “abrirse” en dicha interfaz o una versión posterior. En el caso de utilizar versiones posteriores de la interfaz, i.e. Visual Studio 2008 y .NET 3.5, se deja al lector la tarea de revisar la bitácora de migración con el fin de solucionar cualquier detalle que pudiera surgir; en nuestro caso se mantuvo la implementación en la versión 2.0 de

.NET con el fin de mantener compatibilidad de código con la implementación abierta de .NET (Mono).

Finalmente, en el capítulo 5, tras el desarrollo exitoso del verificador de modelos y la verificación también exitosa de los casos de estudio, presentamos los resultados en forma de tablas de estados. Las tablas que presentamos son extracciones directas de nuestro verificador. Acompañamos las tablas de resultados con las fórmulas que nos permiten obtener dichas tablas.

Las conclusiones alrededor de los resultados que obtuvimos al desarrollar nuestro verificador de modelos pueden encontrarse en el capítulo 6 de este trabajo.

A lo largo de los capítulos presentamos algunos términos en inglés y otros en español; la razón es que en la literatura técnica a veces se emplean palabras en su lenguaje original, i.e. hardware, software, mientras que otras veces se traducen al español, i.e. *ADN*.

CAPÍTULO 2
VERIFICACIÓN DE MODELOS

2.1 Introducción

Los orígenes de la verificación de modelos se remontan a los años 80, cuando Clarke y Emerson y al mismo tiempo pero independientemente Queille y Sifakis introdujeron un nuevo enfoque algorítmico a la verificación de sistemas computacionales. Su idea se basa en la satisfacción de propiedades lógicas en un modelo el cual se representa por una gráfica dirigida. La verificación de modelos ha usado diversas variantes de lógicas temporales como el lenguaje predominante de especificación [1].

Uno de los principales obstáculos a los que se enfrenta cualquier método formal que estudia sistemas de transición es la facilidad con la que crecen dichos sistemas. A este problema se le llama explosión de estados. Un posible enfoque a este problema lo tiene la *verificación de modelos simbólica*. Dicho tipo de verificación utiliza una combinación de algoritmos en conjunto con una estructura de datos llamada *diagramas binarios de decisión* los cuales permiten de una forma eficiente representar conjuntos grandes de estados. En esta área el primer verificador que se utilizó para verificar cientos de variables booleanas fue *SMV* [1].

La verificación de modelos supone sistemas con un número finito de estado, es por ello que naturalmente se enfocó primeramente hacia verificación de hardware, en donde se cumple dicha restricción. Desde los primeros artículos del área, se vio claro que existía una gran motivación por verificar no solamente hardware, sino software.

De [1] extraemos las siguientes palabras y, para preservar su estilo, las colocamos aquí en su lengua original: “Generally though, the principal undecidability of virtually all questions in software verification makes clear that there is no silver bullet for verification, and there will always

be a need to design model checking methods specific to problem classes". Esto sugiere que existen muchas rutas que se pueden emprender para generar nuevas ideas en el área de verificación de modelos, el cual es el tema principal de este trabajo de tesis.

En este capítulo discutimos dos tipos de lógica temporal: la de tipo lineal y la de tipo ramificada. Posteriormente las comparamos y damos los argumentos que nos llevaron a la elección de la lógica ramificada para la implementación de nuestro verificador de modelos. La necesidad de verificar ciertas propiedades sobre redes genéticas nos conduce a extender la lógica seleccionada; esta extensión la cubrimos en la sección de hibridación. Una vez presentadas las bases matemáticas introducimos la verificación de modelos para lo cual mostramos el algoritmo de verificación que utilizamos. Finalmente atacamos el problema de la explosión de estados mediante verificación simbólica; en esta sección presentamos algoritmos para el uso de diagramas binarios de decisión.

2.2 La lógica temporal

En este trabajo nos concierne hablar de sistemas basados en una estructura temporal, más adelante quedará claro por qué. A lo largo del trabajo haremos uso de una lógica temporal en específico. Para poder entender dicha lógica es deseable comenzar describiendo lógica temporal en general.

En la literatura se definen varias lógicas con operadores temporales que nos permiten hablar de propiedades que se cumplen ya sea a lo largo de una trayectoria, o a lo largo de todas las trayectorias.

En general podemos escoger entre dos tipos de lógicas temporales. Una de ellas tiene un enfoque de tipo lineal. En esta lógica el tiempo se identifica como una sucesión lineal de eventos y además se considera que el futuro de todo evento está completamente determinado. La otra lógica se basa (en lugar de en una estructura lineal) en una estructura de tipo ramificada, como un árbol (en el sentido de matemáticas o estructura de datos). De esta manera se permite que un instante en el tiempo tenga más de un sucesor.

En el tipo de lógica que supone trayectorias lineales de estados, se acostumbra adaptar a un sistema con trayectorias ramificadas de estados de la siguiente manera: implícitamente se introduce un cuantificador universal sobre todas las trayectorias lineales abreviadas en un árbol de

trayectorias. De esta manera, esta lógica serviría para caracterizar el conjunto de *todas* las posibles sucesiones de ejecución de un programa a partir de un estado dado y por lo tanto poder verificar ciertas propiedades (que se cumplen o no) en dichas sucesiones.

Por otro lado, la lógica de tipo ramificada considera todos los posibles árboles de ejecución de un programa dado sin introducir necesariamente un cuantificador universal. Sobre un árbol de este tipo podemos analizar propiedades de tipo existencial de al menos una posible ejecución de un programa (podemos considerar a los programas como un sistema no determinístico en el cual algunas de sus posibles ejecuciones lleven a un buen resultado). De forma más general, podemos estudiar propiedades que nos indiquen si existe al menos una ejecución que cumpla con un cierto objetivo. Esto, por supuesto, no implica que todas las posibles ejecuciones cumplan con dicho objetivo [3].

La elección del tipo de lógica debe basarse en el tipo de problema que vayamos a enfrentar. Por razones que se verán más adelante, para este trabajo de tesis resultó conveniente utilizar la lógica temporal que funciona de manera ramificada.

2.3 Linear-time Temporal Logic (LTL)

LTL es una lógica temporal con operadores que nos permiten hacer referencia al futuro. Es una lógica que modela el tiempo como una sucesión de estados los cuales se extienden infinitamente hasta el futuro. A menudo a esta sucesión de estados se le llama también computación, o trayectoria.

En esta lógica trabajamos con un conjunto de fórmulas atómicas o átomos $(p1, p2, p, q, r, \dots)$. Al igual que en otras lógicas, estos átomos son ciertos en un sistema cuando algunos hechos se cumplen, en nuestro caso, en un estado.

Para podernos adentrar un poco más en esta lógica y para poder más adelante adentrarnos en otra lógica presentamos a continuación la sintaxis de *LTL*:

Definición: *LTL* tiene la siguiente sintaxis dada en la forma Backus-Naur (*BNF*) [4]:

$$\phi ::= \mathsf{T} \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (X\phi) \mid (F\phi) \mid (G\phi) \mid (\phi U \phi) \mid (\phi R \phi)$$

En donde p es un átomo tomado de un conjunto *Átomos* de variables proposicionales.

A la izquierda de $(X\phi)$ la sintaxis que se observa es la de la lógica clásica proposicional. A la derecha e incluyendo a $(X\phi)$ se observa la sintaxis de la extensión que hace *LTL* a la lógica proposicional. A los operadores X, F, G, U y R se les llama operadores temporales. A veces se agrega un operador más, W pero para fines de este trabajo no se utilizó:

- X significa *"el siguiente estado",* (en inglés) "*neXt state".*
- F significa *"algún estado en el Futuro o en el presente (Future)".*
- G significa *"todos los estados futuros (Globally)".*
- U *significa "Until"*. Para expresarlo en español, más que usar la traducción literal debería usarse *"Sino… hasta"*; esto se deberá entender mejor cuando se introduzca la semántica de la lógica.
- R significa *"libera (Release)".*

Teniendo definida la sintaxis, podemos identificar algunas fórmulas en *LTL* (sin importarnos, por ahora, lo que significan):

- $Fp \wedge Gp \rightarrow pUr$
- $F(p \rightarrow Gr) \vee \neg qUp$
- $pU(qUr)$
- $GFp \rightarrow F(q \vee s)$

Habiendo cubierto la sintaxis, nos conviene definir ahora la semántica de *LTL*. Para ello habrá que definir antes una estructura, conocida en este tipo de lógicas como *estructura de Kripke o modelo*. Esta estructura será la que represente el sistema en el cual queremos validar ciertas propiedades [5]:

Definición: Estructura $M = \langle S, R, L \rangle$

en donde

S =*Conjunto finito de estados* $\{s_1, s_2, \ldots, s_n\}$

R =*Relación de transición o accesibilidad* $(R \subseteq S^2)$ *y además* R *es total, es decir, que para todo* s*, existe* s' *tal que* $(s, s') \in R$

L =*Función de etiquetamiento* $L: S \rightarrow 2^{Atomos}$ *(es decir, el conjunto potencia del conjunto de variables proposicionales)*

Un ejemplo de este tipo de estructuras sería [4]:

$S = \{s_0, s_1, s_2\}$
$R = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_2), (s_2, s_2)\}$
$L(s_0) = \{p, q\}$
$L(s_1) = \{q, r\}$
$L(s_2) = \{r\}$

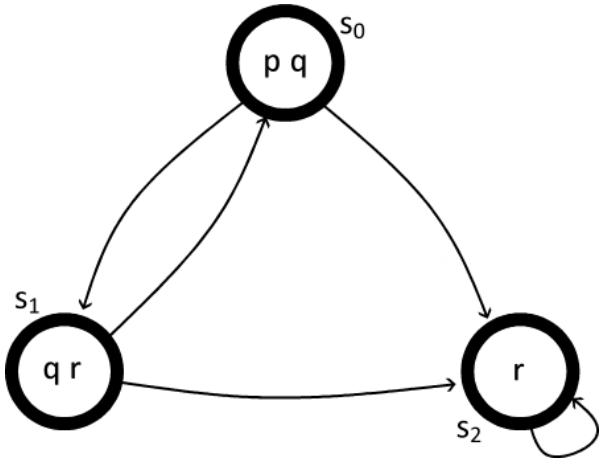


Figura 2.1: Estructura de Kripke o modelo

Para definir la semántica, utilizaremos el concepto de trayectoria del cual ya hemos hablado antes. A una trayectoria la denotaremos mediante π, y diremos que una trayectoria es una sucesión $s_0, s_1, s_2, \ldots$ infinita de estados. Es posible que la trayectoria sea infinita porque la relación de accesibilidad es total. Por ejemplo en la estructura de Kripke dada en la Figura 2.1, una trayectoria sería $\pi = s_0, s_1, s_0, \ldots$ También podemos definir una trayectoria que inicie en el estado s_i de π si escribimos π^i. Por ejemplo $\pi^2 = s_2, \ldots$

La semántica de *LTL* es [4]:

- $\pi \models \top$
- $\pi \not\models \bot$

- $\pi \vDash p$ sii $p \in L(s_0)$
- $\pi \vDash \neg\alpha$ sii $\pi \nvDash \alpha$
- $\pi \vDash \alpha \wedge \beta$ sii $\pi \vDash \alpha$ y $\pi \vDash \beta$
- $\pi \vDash \alpha \vee \beta$ sii $\pi \vDash \alpha$ o $\pi \vDash \beta$
- $\pi \vDash \alpha \rightarrow \beta$ sii $\pi \vDash \beta$ siempre que $\pi \vDash \alpha$
- $\pi \vDash X\alpha$ sii $\pi^1 \vDash \alpha$
- $\pi \vDash F\alpha$ sii existe $i \geq 0$ tal que $\pi^i \vDash \alpha$
- $\pi \vDash G\alpha$ sii para toda $i \geq 0$, $\pi^i \vDash \alpha$
- $\pi \vDash \alpha U\beta$ sii existe $i \geq 0$ tal que $\pi^i \vDash \beta$ y para toda $j = 0, \dots, i-1$, $\pi^j \vDash \alpha$
- $\pi \vDash \alpha R\beta$ sii existe ya sea $i \geq 0$ tal que $\pi^i \vDash \alpha$ y para toda $j = 0, \dots, i$, $\pi^j \vDash \beta$, o para toda $k \geq 0$, $\pi^k \vDash \beta$

Ahora, para poder hacer verificación sobre el modelo especificado anteriormente deberemos sobrecargar el "torniquete" $(\vDash)$ de la siguiente manera:

$M \vDash_s \alpha$

Se cumple sii por cada trayectoria π de M que empieza con s, $\pi \vDash \alpha$.

Aquí observamos ya una característica de *LTL* para verificación de modelos (estructuras de Kripke), pues hay un cuantificador universal implícito en la sobrecarga anterior del torniquete. Por ejemplo, en *LTL* no podemos verificar una cierta propiedad en una sola trayectoria (digamos).

En ocasiones es útil visualizar todas las posibles trayectorias desde un estado dado "desenrollando" el sistema de transiciones (la estructura) para obtener un árbol de trayectorias infinito o árbol de ejecución.

Por ejemplo, si construimos el árbol de ejecución para la estructura en la Figura 2.1 tomando como estado inicial s_0, entonces obtendremos el siguiente árbol infinito [4]:

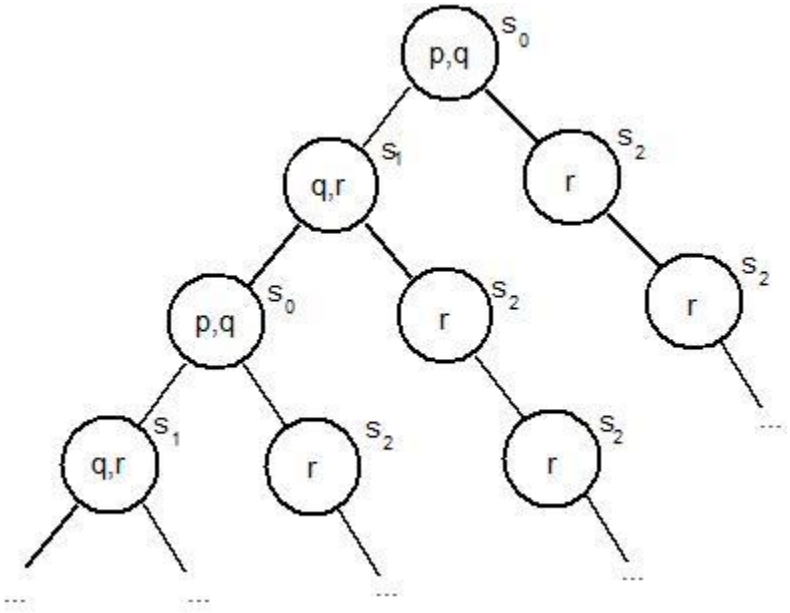


Figura 2.2: "Desenrollado" de la estructura de Kripke presentada en la Figura 2.1

Las trayectorias de un modelo quedan representadas de forma explícita si se utiliza la representación del árbol de ejecución como el presentado en la Figura 2.2. El árbol se puede construir siguiendo los posibles recorridos y conectando los nodos en el árbol conforme se van visitando en la estructura de Kripke. Es posible que el árbol de ejecución sea infinito porque la relación de accesibilidad de la estructura de Kripke en la Figura 2.1 es total.

Ahora podemos verificar si las siguientes propiedades (en *LTL*) se cumplen en la estructura de Kripke antes especificada:

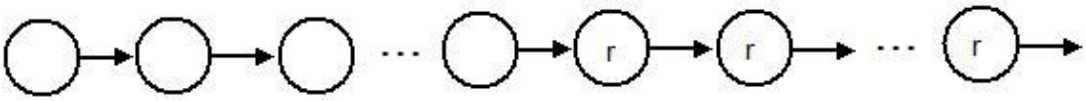
- $M \vDash_{s_0} p \wedge q$ la cual sí se satisface
- $M \vDash_{s_0} \neg r$ la cual sí se satisface

- $M \vDash_{s_0}$T la cual sí se satisface
- $M \vDash_{s_0} Xr$ la cual sí se satisface
- $M \vDash_{s_0} X(q \wedge r)$ la cual no se satisface
- $M \vDash_{s_0} G\neg(p \wedge r)$ la cual sí se satisface

Podemos identificar algunos patrones útiles. Dichos patrones nos servirán para presentar algunos ejemplos más adelante en este capítulo. Los patrones que presentamos a continuación utilizan los operadores *G* y *F*. Resulta interesante que al cambiar el anidamiento de los operadores obtenemos diferente significado para la fórmula.

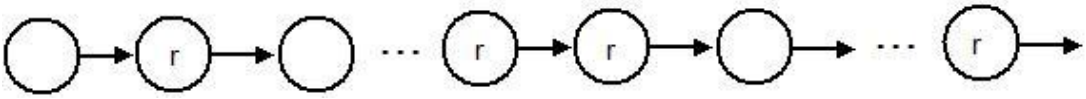
- "Tarde o temprano" tenemos continuamente r. Puede expresarse mediante la fórmula $F\,G\,r$

  El patrón $F\,G\,r$ puede observarse en la siguiente trayectoria:



- "r se cumple de manera infinitamente frecuente". Puede expresarse mediante la fórmula $G\,F\,r$

  El patrón $G\,F\,r$ puede observarse en la siguiente trayectoria:



Existen algunas equivalencias importantes entre fórmulas de *LTL* que vale la pena mencionar. Para ello consideraremos la siguiente definición [4]:

Decimos que dos fórmulas *LTL* α y β son semánticamente equivalentes, o simplemente equivalentes escribiendo $\alpha \equiv \beta$, si para todos los modelos M, y para todas las trayectorias π en $M: \pi \vDash \alpha \; sii \; \pi \vDash \beta$.

La equivalencia entre α y β significa que α y β son semánticamente intercambiables. Si α es una subfórmula de una fórmula más grande Δ, y $\alpha \equiv \beta$, entonces podemos hacer la sustitucióin de α por β en Δ sin cambiar el significado de Δ.

En lógica proposicional, sabemos que $\wedge$ y $\vee$ son duales uno del otro, significando que si utilizamos una negación $\neg$ con $\wedge$, podemos obtener $\vee$ y viceversa:

$$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$$

$$\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

Similarmente, los operadores *LTL* F y G son duales uno del otro, y resulta que X es dual de sí mismo:

$$\neg G\alpha \equiv F\neg\alpha$$

$$\neg F\alpha \equiv G\neg\alpha$$

$$\neg X\alpha \equiv X\neg\alpha$$

También U y R son duales uno del otro:

$$\neg(\alpha U \beta) \equiv \neg\alpha\ R \neg\beta$$

$$\neg(\alpha\ R\ \beta) \equiv \neg\alpha U\ \neg\beta$$

2.4 Computation Tree Logic (CTL)

En la sección anterior de *LTL* (linear-time temporal logic) vimos que las fórmulas se evalúan sobre trayectorias. Se definió que un estado de un sistema satisface una fórmula *LTL* si todas las trayectorias desde el estado dado la satisfacen.

Este problema implica que la existencia de una trayectoria no puede ser expresada con *LTL*. Esto puede ser parcialmente superado considerando la negación de la propiedad en *LTL*. Por ejemplo, para confirmar la existencia de una trayectoria desde *s* que satisfaga la fórmula *LTL* α, podemos revisar si todas las trayectorias satisfacen $\neg\alpha$; una respuesta positiva a esta fórmula es una respuesta negativa a la pregunta original y viceversa.

Sin embargo, las propiedades que combinan cuantificadores universales y existenciales no pueden ser en general verificadas utilizando esta aproximación porque la fórmula complementaria tendrá también una combinación de cuantificadores universales y existenciales.

El tipo de lógicas a la cual pertenece *CTL* resuelven este problema al permitir cuantificadores explícitos sobre trayectorias.

En esta lógica cada uno de los operadores utilizados en *LTL*, como lo son:

- *G*
- *F*
- *X*
- *U*

Se deben escribir con un prefijo que indique un cuantificador sobre trayectorias. En la notación que utilizaremos para *CTL*, se encuentran los siguientes dos cuantificadores sobre trayectorias [2]:

- *A* – Para todas las posibles trayectorias
- *E* – Para alguna posible trayectoria

Las fórmulas en *CTL* se construyen a partir de átomos, operadores booleanos y operadores temporales. Ahora podemos definir la sintaxis para *CTL* [2]:

Definición: *CTL* tiene la siguiente sintaxis expresada en *BNF*:

$$\phi ::= \mathsf{T} \mid \bot \mid p \mid (\neg \phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid$$

$$(EX\phi) \mid (AX\phi) \mid (EF\phi) \mid (AF\phi) \mid (EG\phi) \mid (AG\phi) \mid E[\phi U \phi] \mid A[\phi U \phi]$$

Hay que notar que todos los operadores temporales de *CTL* contienen pares de símbolos. El primer símbolo es el cuantificador sobre trayectorias A y E y el segundo alguno de los operadores X, F, G o U. No podemos escribir fórmulas en *CTL* que incluyan ocurrencias aisladas de A y E o de X, F, G o U.

En esta ocasión será más fácil definir la semántica de *CTL* pues ya hemos definido antes la sobrecarga del torniquete y la estructura representativa del sistema:

- $M \vDash_s \mathsf{T}$
- $M \nvDash_s \bot$
- $M \vDash_s p$ sii $p \in L(s)$
- $M \vDash_s \neg\alpha$ sii $M \nvDash_s \alpha$
- $M \vDash_s \alpha \wedge \beta$ sii $M \vDash_s \alpha$ y $M \vDash_s \beta$
- $M \vDash_s \alpha \vee \beta$ sii $M \vDash_s \alpha$ o $M \vDash_s \beta$
- $M \vDash_s \alpha \rightarrow \beta$ sii $M \vDash_s \beta$ siempre que $M \vDash_s \alpha$
- $M \vDash_s AX\alpha$ sii para toda s' tal que $(s, s') \in R, M \vDash_{s'} \alpha$
- $M \vDash_s EX\alpha$ sii para alguna s' tal que $(s, s') \in R, M \vDash_{s'} \alpha$
- $M \vDash_s AF\alpha$ sii para toda trayectoria $(s_0, s_1, s_2, \ldots)$, $M \vDash_{s_i} \alpha$ para alguna i
- $M \vDash_s EF\alpha$ sii para alguna trayectoria $(s_0, s_1, s_2, \ldots)$, $M \vDash_{s_i} \alpha$ para alguna i
- $M \vDash_s AG\alpha$ sii para toda trayectoria $(s_0, s_1, s_2, \ldots)$, $M \vDash_{s_i} \alpha$ para toda i
- $M \vDash_s EG\alpha$ sii para alguna trayectoria $(s_0, s_1, s_2, \ldots)$, $M \vDash_{s_i} \alpha$ para toda i
- $M \vDash_s A(\alpha U\beta)$ sii para toda trayectoria $(s_0, s_1, s_2, \ldots) = \pi$, $\pi \vDash \alpha U\beta$ (LTL)
- $M \vDash_s E(\alpha U\beta)$ sii para alguna trayectoria $(s_0, s_1, s_2, \ldots) = \pi$, $\pi \vDash \alpha U\beta$ (LTL)

Podemos entonces definir algunas propiedades en *CTL* dada la estructura (Figura 1) definida anteriormente, y verificar cuáles se satisfacen en M:

- $M \vDash_{s_0} p \wedge q$, sí se satisface en M
- $M \vDash_{s_0} \neg r$, sí se satisface en M
- $M \vDash_{s_0} EX(q \wedge r)$, sí se satisface en M
- $M \vDash_{s_0} \neg AX(q \wedge r)$, sí se satisface en M
- $M \vDash_{s_0} \neg EF(p \wedge r)$, sí se satisface en M
- $M \vDash_{s_0} A(pUr)$, sí se satisface en M
- $M \vDash_{s_0} E\big((p \wedge r)Ur\big)$, no se satisface en M

La dualidad de los cuantificadores A y E nos permite distinguir propiedades que *se deben* cumplir en todas las posibles trayectorias, y cuantificadores que *pueden* cumplirse en algunas trayectorias.

Al igual que con *LTL*, existen algunas fórmulas equivalentes que vale la pena mencionar. Para esto utilizaremos la siguiente definición [4]:

Decimos que dos fórmulas *CTL* α y β son semánticamente equivalentes, o simplemente equivalentes escribiendo $\alpha \equiv \beta$, si cualquier estado en cualquier modelo que satisface a una de ellas también satisface a la otra.

Podemos expresar las reglas similares a las de De Morgan:

$\neg AF\alpha \equiv EG\neg\alpha$

$\neg EF\alpha \equiv AG\neg\alpha$

$\neg AX\alpha \equiv EX\neg\alpha$

También tenemos las equivalencias:

$$AF\alpha \equiv A[\top U\ \alpha]$$

$$EF\alpha \equiv E[\top U\ \alpha]$$

Si consideramos las equivalencias presentadas para *LTL*, podemos definir el operador R de la siguiente manera:

$$A[\alpha R\beta] = \neg E[\neg\alpha U\neg\beta]$$

$$E[\alpha R\beta] = \neg A[\neg\alpha U\neg\beta]$$

Al igual que se hizo en la sección de *LTL*, para visualizar las posibles trayectorias a partir de un estado cualquiera resulta útil desenrollar la estructura y construir el árbol de ejecución que las representa.

En esta ocasión los árboles de ejecución servirán para visualizar estructuras cuyo estado inicial satisface algunas de las fórmulas antes expuestas para *CTL*.

La fórmula $EF\ r$, que se puede leer como "existe alguna trayectoria en la que tarde o temprano cumple r" se puede visualizar fácilmente si utilizamos la representación del árbol de ejecución:

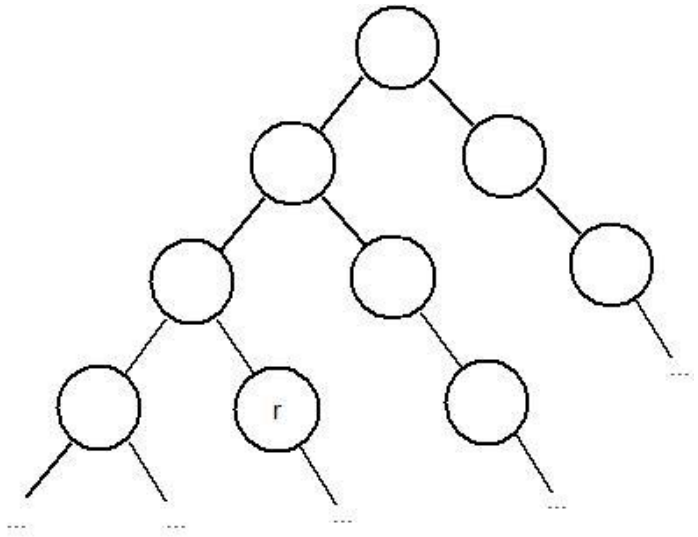


Figura 2.3: Árbol de ejecución para una estructura cuyo estado inicial satisface $EF\ r$

Ahora es más claro visualizar el significado de la fórmula en la Figura 2.3. Para que $EF\ r$ sea satisfecha basta con que en alguna de las trayectorias que resultan del estado inicial de evaluación se cumpla r.

En los árboles de ejecución que se presentan a continuación puede apreciarse con mayor claridad la diferencia entre los cuantificadores E y A.

La fórmula $EG\ r$ puede leerse como "existe alguna trayectoria en la que siempre se cumple r":

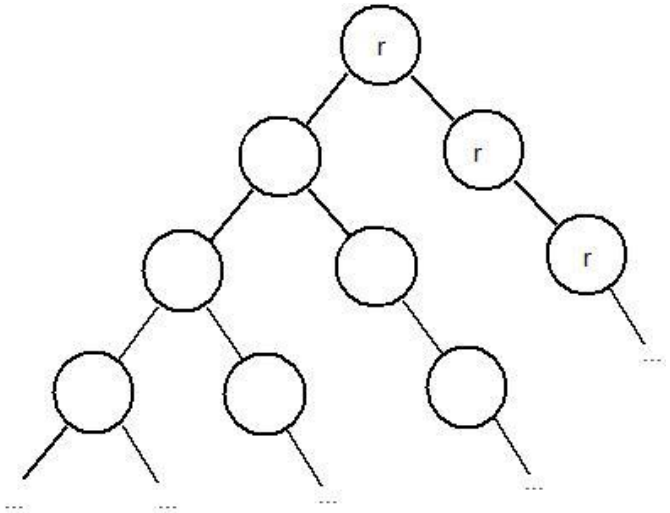


Figura 2.4: Árbol de ejecución para una estructura cuyo estado inicial satisface $EG\ r$

Hay que resaltar que para que $EG\ r$ sea satisfecha en una estructura dada, r debe cumplirse desde el estado inicial de evaluación y en una trayectoria completa (puede cumplirse en varias e incluso en todas).

Ahora la diferencia con el cuantificador universal queda clara cuando observamos el siguiente árbol de ejecución para la fórmula $AG\ r$, la cual puede leerse como "en todas las trayectorias hasta el infinito se cumple r":

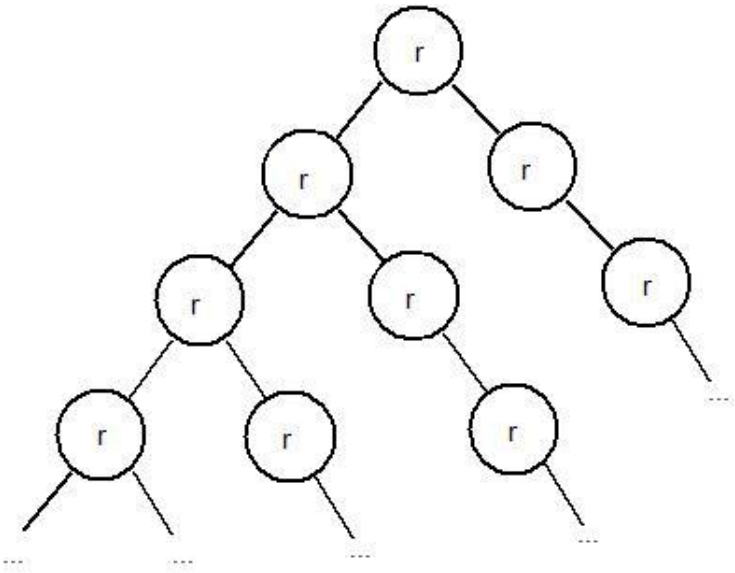


Figura 2.5: Árbol de ejecución para una estructura cuyo estado inicial satisface $AG\ r$

Al igual que con $EG\ r$, la fórmula $AG\ r$ debe cumplir r desde el estado inicial de evaluación.

Con una idea más clara sobre la diferencia entre los cuantificadores, se puede presentar en la Figura 2.6 el árbol de ejecución para la fórmula $AF\ r$ la cual puede leerse como "en toda trayectoria tarde o temprano se cumple r".

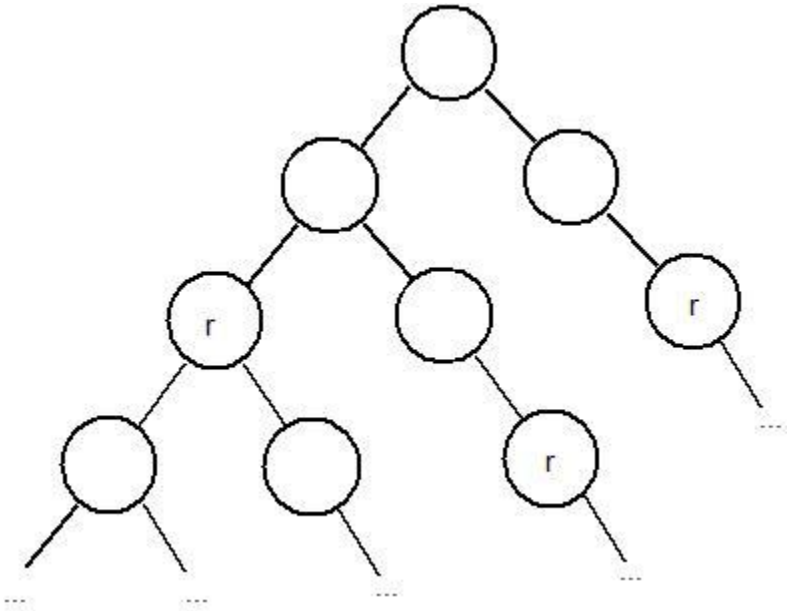


Figura 2.6: Árbol de ejecución para una estructura cuyo estado inicial satisface $AF\ r$

En este árbol de ejecución puede observarse que para $AF\ r$ si se cumple r en un nodo raíz; ya no es necesario que se cumpla en sus hijos para que la fórmula se satisfaga.

2.5 LTL vs CTL

La ventaja de una lógica sobre la otra no es clara. Vamos a ver que ninguna es más expresiva que la otra.

Resulta útil ver algunos ejemplos de fórmulas y compararlas entre ambas lógicas.

Por ejemplo, en *LTL* podemos seleccionar trayectorias haciendo uso de la implicación [4]:

- *Todas las trayectorias que "tienen" p también tienen q*
  $Fp \rightarrow Fq$
- En *CTL* podríamos tratar de aproximarlo mediante la siguiente propiedad:

$AFp \rightarrow AFq$

Sin embargo, esta propiedad expresa algo diferente: *Si todas las trayectorias tienen p entonces todas tienen q.*

Podemos escribir otro ejemplo de una propiedad que puede ser solamente expresada en *LTL*, pero no en *CTL*; para ello usaremos un ejemplo con contenido mundano:

- *Si un proceso se habilita de manera infinitamente frecuente, entonces corre de manera infinitamente frecuente.*
  En LTL: $GF\ habilitado \rightarrow GF\ corriendo$
  Y en CTL esto no es expresable.
  Si tratamos de aproximarlo con $AG\,AF\ habilitado \rightarrow AG\,AF\ corriendo$
  Estaremos expresando algo un tanto diferente: si toda trayectoria tiene habilitado de manera infinitamente frecuente entonces toda trayectoria tiene corriendo de manera infinitamente frecuente. Esto claramente no expresa lo mismo que el enunciado original.

Concluimos que el poder expresivo de ambas lógicas resulta incomparable. Podemos expresar cosas muy diferentes con cada lógica lo cual complica la decisión sobre cuál es preferible en cuanto a expresividad y potencial.

CTL, como puede observarse en las Figuras 2.3-2.6, tiene la gran ventaja de poder cuantificar sobre trayectorias. Sin embargo, podemos observar en *LTL* la posibilidad de detallar propiedades sobre trayectorias individuales.

En nuestra opinión, intuitivamente *LTL* es una lógica que resulta más fácil de entender. Sin embargo, teóricamente la verificación es computacionalmente más sencilla con *CTL*. La verificación con *CTL* puede hacerse en tiempo lineal tanto en el tamaño de la estructura como en el de la fórmula, mientras que la verificación con *LTL* puede tomar tiempo exponencial en el tamaño de la fórmula. A pesar de esto, en la práctica la verificación para *LTL* puede resultar tan eficiente como la de *CTL* porque en general la fórmula es pequeña con respecto del modelo [8].

La ventaja teórica en la complejidad de *CTL* sobre *LTL* en la verificación de modelos hace que *CTL* sea una opción común, resultando en herramientas para verificación eficientes. Algunas de las herramientas que han surgido para verificación de modelos son *SMV*, el primer verificador de

modelos simbólico, el cual está basado en *CTL*, uno de sus sucesores *VIS*, también basado en *CTL*, lo cual indica que la lógica temporal ramificada es la principal elección a la hora de hacer verificación de modelos.

Sin embargo, aún con el aparente éxito que *CTL* ha tenido en la verificación de modelos, tiene ciertas limitantes importantes como lenguaje de especificación en cuanto a expresividad; todo esto derivado de que *CTL* es un lenguaje en una lógica temporal ramificada. Además resulta ser un lenguaje de especificación más difícil de entender.

La expresividad no se reduce únicamente a un problema teórico; tiene mucho que ver con un problema de usabilidad. En [8] se menciona que los ingenieros que se dedican a verificación encuentran que *CTL* es poco intuitivo. El contexto lineal es simplemente más natural, la gente tiende a pensar de forma lineal. Por ejemplo, se suelen utilizar diagramas de tiempo, gráficas de secuencias, etc.

Para nuestros casos de estudio, creemos que *CTL* tiene dos ventajas importantes sobre *LTL*. Por un lado, una lógica de tipo ramificada como *CTL* nos permite diferenciar entre propiedades que se *deben* cumplir y propiedades que se *pueden* cumplir; algunas de las propiedades de interés en redes genéticas requieren dicha diferencia [21]. Por otro lado, podemos utilizar *CTL* para calcular los estados que cumplen con una cierta propiedad; con *LTL* solo podemos calcular una trayectoria (el contraejemplo).

La elección de una lógica temporal (*LTL* o *CTL*) resulta más bien de la aplicación que se quiera desarrollar, y por supuesto tendrá mucho que ver con preferencias personales y desde luego con el entendimiento que se logre de una u otra lógica.

Para los modelos que se buscan verificar en este proyecto de tesis, se desea validar si ciertas propiedades se encuentran o no presentes sin la necesidad de detallar trayectorias. Además, es importante poder lidiar con incertidumbre en la estructura. La posibilidad de especificar propiedades que se pueden cumplir en algunas trayectorias también es muy importante. Es por ello que la elección fue finalmente utilizar *CTL*.

Aún con todas las ventajas y desventajas que puede tener utilizar *CTL*, la verificación sufre de un problema importante: la explosión de estados. Esto quiere decir que la complejidad de analizar estructuras con muchas variables se convierte en un serio problema computacional. De ahí que

surge la necesidad de extender la verificación hacia una representación "simbólica" del modelo, lo cual será tratado más adelante en este mismo capítulo.

Algunas de las propiedades deseables en una red genética no son expresables con fórmulas en *CTL* (tampoco con fórmulas en *LTL*) ya que resulta necesario poder referirse a estados de alguna manera. Esto, por el contrario, sí es posible si se utiliza una lógica como es la de primer orden. Por lo tanto, resulta inevitable tratar de incluir cierta nueva expresividad a la lógica de especificación (en nuestro caso *CTL*) lo cual nos lleva a la siguiente sección, en la cual se discute el tema de lógicas híbridas y se particulariza para la lógica ramificada *CTL*.

2.6 Hibridación de CTL

2.6.1 Historia de la lógica híbrida

Esta sección se basa principalmente en [7] y [M1].

Arthur Prior es considerado uno de los más importantes promotores de las lógicas modales. Prior, en colaboración con Carew Meredith, elaboró una versión de una semántica para mundos posibles. Al mismo tiempo pero de forma independiente, Carnap trabajó en semánticas modales. Ambas investigaciones se hicieron antes que Kripke publicara su primer artículo del tema.

En 1954, en el Congreso de Filosofía en Nueva Zelanda, Prior presentó un artículo en el cual introdujo el *cálculo I* (el cual después llamaría *cálculo U*). En el *cálculo I*, las proposiciones son tratadas como predicados que expresan propiedades de fechas (que se representan mediante variables).

Las ideas de Prior fueron más tarde retomadas por su estudiante Robert Bull. Las ideas de Bull fueron más allá, a tal grado que definió una forma para poder nombrar trayectorias en un modelo.

El siguiente paso fue dado por la Escuela de Sofía en Bulgaria, la cual de forma independiente reiventó la idea de "nominales" y lenguajes híbridos durante la década de 1980. Gargov, Passy y Tinchev son algunos de los investigadores de la Escuela de Sofía que trabajaron lógicas modales.

Hoy en día, la lógica modal se puede ver como un fragmento de lógica de primer orden o de segundo orden. Esto resulta casi obvio cuando observamos la semántica para mundos posibles o los operadores modales.

2.6.2 ¿Qué es la lógica híbrida?

Algo necesario para la aplicación de nuestro verificador, y que con *CTL* simple no se puede conseguir es nombrar estados, y por lo tanto llevar a cabo una verificación sobre ellos, es decir, poder construir fórmulas que permitan especificar estados como parte de la propiedad que se busca verificar. Con esta idea surge la necesidad de extender de cierta manera *CTL* para poder incorporar algún tipo de variables y cuantificadores que permitan especificar propiedades/fórmulas en términos de estados.

Esto sugiere que la lógica temporal elegida (*CTL*) debe extenderse con algunos elementos de lógica de primer orden. Las lógicas híbridas son lógicas modales que tienen símbolos especiales para nombrar estados individuales dentro del modelo. Estos nuevos símbolos, que a menudo son llamados *nominales*, se agregan como nuevos símbolos atómicos. Una vez agregados a la lógica, podemos combinarlos con las variables proposicionales para construir nuevas fórmulas [7].

Un nominal denota un conjunto singular de estados. Como ya se mencionó, un nominal sirve para nombrar un estado individual. Esto quiere decir que son verdaderos únicamente en un punto dentro del modelo (i.e. en un estado). Esto tiene algunas repercusiones importantes. Por ejemplo, si dos fórmulas son verdaderas dado un modelo M y un estado m, y ambas fórmulas incluyen un mismo nominal i, entonces no importa cuántas variables proposicionales hayamos puesto además del nominal, sabemos que ambas fórmulas se validan en el mismo estado m'.

Esto queda más claro con el siguiente ejemplo extraído de [7], supónganse por ahora los operadores temporales de *LTL*:

Sea M un modelo, m un estado en el dominio de M, y suponga $M, m \vDash X(i \wedge p) \wedge X(i \wedge q)$. Sea m' un estado sucesor de m que satisface $i \wedge p$, y m'' otro estado sucesor de m el cual satisface $i \wedge q$. Como i es un nominal, y sabemos que es verdadero en un único punto dentro del modelo, entonces podemos concluir que $m' = m''$, y por lo tanto tenemos $M, m \vDash X(p \wedge q)$.

La definición de los nominales permite sugerir el operador $@_i$ por cada nominal en el modelo. El operador $@$ nos permite saltar al punto llamado *i* dentro del modelo con el cual estemos trabajando. Después de todo si podemos ponerle nombre a los estados, entonces conviene poder "ir" a ellos de alguna forma. Por ejemplo, la fórmula $@_i\alpha$ mueve el punto de evaluación al estado llamado *i* y evalúa α en esa posición [7].

Junto con la posibilidad de nombrar estados de forma individual, viene la idea de flexibilizar un poco más la lógica, y de esta manera poder también cuantificar estados. Uno de los operadores que más se utiliza en las lógicas híbridas es el ligador ("binder" en inglés) $\downarrow$, el cual nos permite ligar una variable a puntos dentro del modelo (estados).

En otras palabras, el ligador nos permite ponerle nombre al "aquí y ahora" y de esta manera poder referirnos a él después dentro de una fórmula.

Por ejemplo, la fórmula:

$$\downarrow y.X(y)$$

se valida en un estado m sii m tiene una relación consigo mismo. La fórmula anterior podría leerse de la siguiente manera: "invoca al estado actual y y verifica si y es alcanzable en exactamente una transición".

2.6.3 Especificación de la extensión para CTL

La lógica híbrida extiende la lógica temporal ramificada *CTL* con artefactos que nos permiten nombrar estados y nos brinda la posibilidad de acceder a ellos mediante sus nombres.

Para poder definir la especificación de esta extensión para *CTL*, consideremos lo siguiente:

Sea S el conjunto de todos los estados. Agregamos $VAR = \{x, y, z, \ldots\}$ un conjunto de variables (variables de estado) que toman valores sobre S.

Entonces podemos definir la sintaxis en *BNF* para la hibridación de *CTL* como se muestra a continuación:

$\phi ::= CTL \mid s \mid x \mid @_s\alpha \mid @_x\alpha \mid \downarrow x.\alpha \mid \exists x.\alpha$

En donde $s \in S$ y $x \in VAR$.

Siguiendo la metodología empleada para *LTL* y *CTL*, a continuación se presenta la semántica para la hibridación propuesta:

- $M \vDash_s s'$ sii $s = s'$
- $M \vDash_s @_{s'}\alpha$ sii $M \vDash_{s'} \alpha$
- $M \vDash_s \downarrow x.\alpha$ sii $M \vDash_s \alpha[s/x]$ en donde s/x significa reemplazar todas las ocurrencias de x por s.
- $M \vDash_s \exists x.\alpha$ sii existe $s' \in S$ tal que $M \vDash_s \alpha[s'/x]$

Para la definición de la semántica, no es necesario definir los casos para x ya que, debido a la sustitución $[s/x]$ del ligador, este se reduce al caso de s y tampoco es necesario especificar el caso para $@_x\alpha$ ya que este se reduce al caso de $@_s\alpha$.

Puesto en palabras, el operador $@_s$ mueve la evaluación a s, tal que $s \in S$ o $s \in VAR$, es decir, no importando el estado en el cual la evaluación se esté llevando a cabo, se realiza un salto (como el "goto" de algunos lenguajes de programación) hacia el estado especificado y se evalúa el resto de la fórmula en el nuevo estado.

El operador ligador $\downarrow x$, liga la variable de estado x al estado actual (en donde se esté llevando la evaluación), tal que $x \in VAR$. De esta manera en lo posterior puede utilizarse x dentro de la fórmula siendo evaluada y se estará haciendo referencia al estado al cual se hizo la ligadura.

El operador existencial $\exists x$ liga la variable de estado x a algún estado de la estructura siendo evaluada.

Los operadores $\downarrow$ y $\exists$ no mueven el punto de evaluación del estado actual.

2.7 Verificación de modelos

Esta sección se basa principalmente en [4].

La verificación de modelos es un proceso que consiste en determinar si una fórmula cualquiera α en una lógica particular es satisfecha en un modelo M y estado s arbitrarios. Tanto el modelo M como el estado s y la fórmula α que se busca satisfacer (que podría verse como una hipótesis que se tiene sobre M), son dados por el usuario.

La verificación de modelos se basa en lógicas temporales. La idea principal de las lógicas temporales es que una fórmula no es estáticamente satisfecha o no satisfecha en un modelo, como sería en el caso de la lógica proposicional y de predicados. En cambio, los modelos de lógicas temporales pueden contener muchos estados y una fórmula puede ser satisfecha en algunos estados y no ser satisfecha en otros. De esta manera, la noción de satisfacción se reemplaza por una noción dinámica de satisfacción, en donde las fórmulas pueden cambiar sus valores de satisfacción conforme el sistema evoluciona de estado en estado.

En la verificación de modelos, los modelos M representan sistemas de transiciones y las fórmulas α representan propiedades.

Para verificar que un sistema satisface una propiedad, debemos hacer básicamente tres cosas:

- Especificar el sistema a verificar usando el lenguaje de descripción de acuerdo con el verificador de modelos, para así poder obtener M (usamos “tablas” que explicaremos en el capítulo 4).

- Escribir la propiedad usando el lenguaje de especificación del verificador, para así poder obtener la fórmula α (usamos *CTL* híbrido con algunos símbolos cambiados para que sea posible emplear un teclado estándar).
- Ejecutar el verificador con M y α como entradas.

A partir de esto, un verificador tradicional arrojaría una respuesta afirmativa o negativa dependiendo si la fórmula se cumple o no en el modelo. En el caso del verificador desarrollado en esta tesis, la respuesta afirmativa no se reduce únicamente a un "sí" sino que se imprime el conjunto de estados que satisfacen α.

2.7.1 Algoritmo de verificación

En esta sección se describe el algoritmo de etiquetamiento presentado en [4] páginas 225-229.

El algoritmo de etiquetamiento recibe como entradas una fórmula especificada en *CTL* así como un modelo M que representa los estados y las transiciones entre ellos y arroja como salida el conjunto de estados que satisfacen la fórmula especificada.

Además, para la especificación del algoritmo se considera lo siguiente:

- Existe cierta redundancia entre los operadores en *CTL*.

  Por ejemplo, $AX\alpha$ puede escribirse como $\neg EX\neg\alpha$; los operadores AG, AF, EG y EF se pueden escribir en términos de AU y EU de la siguiente manera (tomar en cuenta las equivalencias presentadas en la sección 4 de este capítulo):

  $AG\alpha$ puede escribirse como $\neg EF\neg\alpha$ y $EG\alpha$ como $\neg AF\neg\alpha$.

  De esta manera podemos decir que AU, EU y EX forman un *conjunto adecuado de operadores temporales*.

La definición de conjunto adecuado es: Un conjunto mínimo de operadores en términos de los cuales se pueden expresar todos los demás.

En base a la definición anterior, tenemos el siguiente teorema: Un conjunto de operadores temporales en *CTL* es adecuado si, y solo si, este contiene al menos un operador de $\{AX, EX\}$, al menos uno de $\{EG, AF, AU\}$ y EU.

Esto nos permite definir el algoritmo en términos de tres primitivas, las primitivas serán entonces el *conjunto adecuado de operadores temporales* que se haya elegido.

Para el algoritmo de etiquetamiento utilizado en este trabajo consideraremos AF, EU y EX como el conjunto adecuado para la parte temporal, y consideraremos $\bot$, $\neg$ y $\wedge$ como el conjunto adecuado para los operadores no temporales.

De lo anterior, podemos decir que dada una fórmula α, podemos simplemente pre-procesarla de tal manera que la podamos verificar en términos de una representación equivalente utilizando únicamente operadores pertenecientes al conjunto adecuado.

El algoritmo de etiquetamiento visto de forma general quedaría como se muestra a continuación:

ENTRADA: Un modelo M y una fórmula α también en *CTL*.

SALIDA: El conjunto de estados de M que satisface α.

$\alpha \leftarrow PRE\_PROCESAR(\alpha)$

Etiquetar los estados en M con las subfórmulas de α que sean satisfechas en dichos estados, comenzando por las subfórmulas internas y poco a poco saliendo hacia las externas.

Para aclarar el algoritmo anterior, supongamos que β es una subfórmula de α y que los estados que satisfacen todas las subfórmlas inmediatas de β ya fueron etiquetados. Entonces podemos determinar mediante un análisis de casos los estados que debemos etiquetar con β de la siguiente manera:

Si β es:

- $\bot$: No se etiqueta estado alguno.
- p: etiquetar s con p sii $p \in L(s)$ en donde L es la función de etiquetamiento definida en M.

- $\beta_1 \wedge \beta_2$: etiquetar s con $\beta_1 \wedge \beta_2$ si s ya fue etiquetado tanto con β_1 como con β_2.
- $\neg\beta_1$: etiquetar s con $\neg\beta_1$ si s no ha sido etiquetado con β_1.
- $AF\beta_1$:
  - Etiquetar con $AF\beta_1$ cualquier estado s que haya sido etiquetado con β_1.
  - Repetir: etiquetar cualquier estado con $AF\beta_1$ si todos los estados sucesores han sido etiquetados con $AF\beta_1$ hasta que no haya cambios.
- $E[\beta_1\ U\ \beta_2]$:
  - Etiquetar con $E[\beta_1\ U\ \beta_2]$ cualquier estado s que haya sido etiquetado con β_2.
  - Repetir: etiquetar cualquier estado con $E[\beta_1\ U\ \beta_2]$ si éste ha sido etiquetado con β_1 y al menos uno de sus sucesores ha sido etiquetado con $E[\beta_1\ U\ \beta_2]$, hasta que no haya cambios.
- $EX\beta_1$: etiquetar con $EX\beta_1$ cualquier estado si alguno de sus sucesores ha sido etiquetado con β_1.

Finalmente, habiendo etiquetado todos los estados con todas las subfórmulas de α incluida la propia α, lo que se arroja como salida es el conjunto de estados etiquetados con α.

A continuación se presenta el pseudo-código para el algoritmo de etiquetamiento utilizado en este trabajo. La función principal $check$ toma como entrada una fórmula en *CTL*.

El pseudo-código utiliza ciertas palabras como *return*, *local var*, *repeat until*, los cuales indican respectivamente el resultado que arroja la función, la declaración de una variable local y la ejecución de una serie de sentencias de forma repetida hasta que la condición se cumpla.

La traducción o pre-procesamiento de la fórmula puede observarse a lo largo del pseudo-código que se presenta a continuación:

```
function check(α)
begin
    case
        α es T: return S
        α es ⊥: return Ø
        α es un átomo: return {s ∈ S | α ∈ L(s)}
```

α es $\neg\alpha_1$: return $S - check(\alpha_1)$
α es $\alpha_1 \wedge \alpha_2$: return $check(\alpha_1) \cap check(\alpha_2)$
α es $\alpha_1 \vee \alpha_2$: return $check(\alpha_1) \cup check(\alpha_2)$
α es $\alpha_1 \rightarrow \alpha_2$: return $check(\neg\alpha_1 \vee \alpha_2)$
α es $AX(\alpha_1)$: return $check(\neg EX \neg\alpha_1)$
α es $EX(\alpha_1)$: return $check_{EX}(\alpha_1)$
α es $A[\alpha_1\ U\ \alpha_2]$: return $check\big(\neg(E[\neg\alpha_2\ U\ (\neg\alpha_1 \wedge \neg\alpha_2)] \vee EG\neg\alpha_2)\big)$
α es $E[\alpha_1\ U\ \alpha_2]$: return $check_{EU}(\alpha_1, \alpha_2)$
α es $EF\alpha_1$: return $check(E(\mathrm{T}U\alpha_1))$
α es $EG\alpha_1$: return $check(\neg AF \neg\alpha_1)$
α es $AF\alpha_1$: return $check_{AF}(\alpha_1)$
α es $AG\alpha_1$: return $check(\neg EF \neg\alpha_1)$

end case

end function

Como puede observarse, el algoritmo de etiquetamiento utiliza tres funciones auxiliares, $check_{EX}$, $check_{EU}$, y $check_{AF}$. Como se verá a continuación estas funciones auxiliares pueden a su vez hacer llamadas sobre la función principal $check$.

Además se considera como precondición que el algoritmo tiene acceso total al modelo M.

Estas funciones auxiliares hacen uso de dos primitivas:

- $pre_{\exists}(Y) = \{s \in S \mid existe\ s',\ (s \rightarrow s'\ y\ s' \in Y)\}$
- $pre_{\forall}(Y) = \{s \in S \mid existe\ s',\ (s \rightarrow s'\ implica\ s' \in Y)\}$

En donde pre significa la pre-imagen del conjunto de estados.

La primitiva $pre_{\exists}$ toma un conjunto Y de estados y regresa el conjunto de estados que pueden hacer una transición hacia Y. La primitiva $pre_{\forall}$, toma un conjunto Y de estados y regresa un conjunto de estados que hacen transición únicamente hacia Y.

La primitiva $pre_{\forall}$ se puede expresar en términos de $pre_{\exists}$ de la siguiente forma:

$$pre_{\forall}(Y) = S - pre_{\exists}(S - Y)$$

En donde $S - Y$ es el conjunto de todos los estados $s \in S$ y que no se encuentran en Y.

El pseudo-código para las funciones auxiliares es el siguiente.

$check_{EX}$ obtiene los estados que satisfacen α_1 haciendo uso de la función principal $check$ y la primitiva $pre_{\exists}$:

$function\ check_{EX}(\alpha_1)$
$begin$
    return $(pre_{\exists}(check(\alpha_1)))$
$end\ function$

$check_{AF}$ obtiene los estados que satisfacen α_1 haciendo uso de la función principal $check$ y la primitiva $pre_{\forall}$:

$function\ check_{AF}(\alpha_1)$
$local\ var\ X, Y$
$begin$
    $X := S$
    $Y := check(\alpha_1)$
    $while\ X \neq Y\ do$
        $X := Y$
        $Y := Y \cup pre_{\forall}(Y)$
    $end\ while$
    return y
$end\ function$

$check_{EU}$ obtiene los estados que satisfacen α_1 haciendo uso de la función principal $check$ y la primitiva $pre_{\exists}$:

$function\ check_{EU}(\alpha_1, \alpha_2)$

$local\ var\ X, Y, Z$

$begin$

$Z := check(\alpha_1)$

$X := S$

$Y := check(\alpha_2)$

$while\ X \neq Y\ do$

$X := Y$

$Y := Y \cup (Z \cap pre_{\exists}(Y))$

$end\ while$

return Y

$end\ function$

La complejidad del algoritmo presentado es $O(f \cdot V \cdot (V + E))$, en donde f es el número de operadores en la fórmula α, $V = |S|$ i.e. el número de estados y $E = |R|$ i.e. el número de transiciones.

Esto significa que el algoritmo de etiquetamiento utilizado en este trabajo es lineal en el tamaño de la fórmula pero cuadrático en el tamaño del modelo a verificar. Podríamos lograr un algoritmo de etiquetamiento lineal en el tamaño del modelo si intercambiamos el operador AF por EG como parte del conjunto adecuado. Para los operadores EX y EU tendríamos que tomar en cuenta no pasar más de una vez por cada nodo del modelo. Para el caso de EG sería necesario implementar el algoritmo de Tarjan para componentes fuertemente conexas (*CFC*); las *CFC* son regiones del espacio de estados en donde cada estado se encuentra ligado (existe una trayectoria finita a) con todos los estados en esa misma región. La complejidad en dichas condiciones sería $O(f \cdot (V + E))$.

2.8 Verificación simbólica

Esta sección se basa principalmente en [2] y [9].

Durante los últimos años, se ha convertido en evidente que los sistemas de estados finitos pueden ser verificados automáticamente examinando la gráfica de estados que modela el

comportamiento del sistema. Varios métodos han sido propuestos para cumplir dicho cometido; entre ellos se encuentra la verificación de modelos con lógicas temporales. Los métodos propuestos utilizan diferentes modelos y diferentes nociones de verificación. Dado que el número de estados en el modelo puede crecer de forma exponencial, se convierte este crecimiento en uno de los principales problemas computacionales al momento de realizar una verificación automática.

La técnica empleada en este trabajo para combatir la explosión de estados en los sistemas, es representar los estados de forma simbólica en lugar de hacerlo de forma explícita.

A menudo los sistemas reales con un gran número de estados suelen tener una estructura regular lo cual sugiere que exista la misma regularidad en la gráfica de los estados. Por lo tanto, es posible encontrar una mejor forma de representación que permita aprovechar dicha regularidad que las listas o tablas no pueden aprovechar. Un buen candidato para dicha representación simbólica son los *diagramas binarios de decisión* (*BDD* por sus siglas en inglés), los cuales han sido ampliamente usados en diferentes herramientas para el diseño y análisis de circuitos digitales.

Los *BDDs* permiten verificar sistemas con una gran cantidad de estados, sistemas que sería imposible manejar mediante mecanismos de enumeración explícita de estados. Con los *BDDs* es posible verificar modelos con tantos estados como 10^{20}, mientras que con un mecanismo de enumeración explícita estaríamos limitados a un número no mayor que 10^{8} estados. Por lo tanto queda claro que la diferencia es enorme y que la verificación de modelos simbólica representa un paradigma totalmente diferente a la verificación explícita [9].

2.8.1 Diagramas Binarios de Decisión

Los *BDD* son una forma para representar funciones booleanas. Una clase de *BDD* será la que se utilice para el desarrollo del algoritmo de verificación simbólica utilizado en este trabajo.

Los diagramas binarios de decisión fueron considerados en un principio como una estructura más simple llamada *árboles binarios de decisión*. Los árboles binarios de decisión son árboles cuyos nodos no terminales son etiquetados con variables booleanas x, y, z, … y cuyos nodos terminales son etiquetados ya sea con 0 o 1. Cada nodo no terminal tiene dos aristas, una representada comúnmente mediante una línea punteada y otra con una línea sólida. La siguiente figura es un ejemplo de este tipo de árboles con únicamente dos etiquetas x y y:

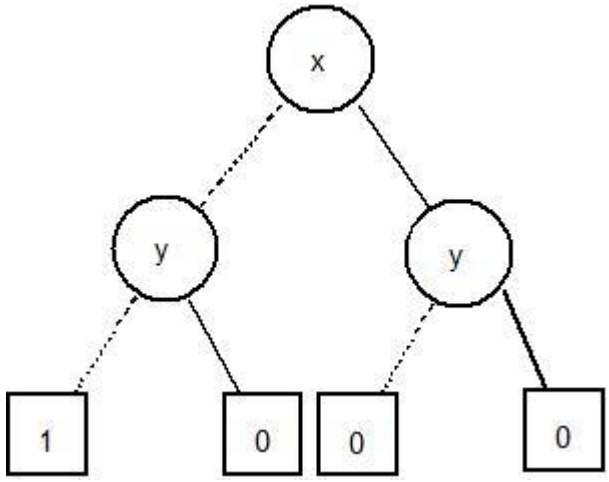


Figura 2.7: Árbol binario de decisión

Para poder emplear estas estructuras podemos seguir la siguiente definición: Sea T un árbol binario de decisión finito. Entonces T determina una función booleana única a partir de las variables en los nodos no terminales de la siguiente manera. Dada una asignación de ceros y unos a las variables booleanas en T, comenzamos desde la raíz de T y seguimos la línea punteada cuando el valor de la variable en el nodo actual sea cero, en otro caso seguimos la línea sólida. El valor de la función es el valor del nodo terminal al cual lleguemos.

Por ejemplo, el árbol binario de decisión de la Figura 2.7 representa la función $f(x,y)$. Para encontrar el valor de la función cuando $x = 0$ y $y = 1$, es decir $f(0,1)$, hacemos lo siguiente a partir de la raíz del árbol:

Como el valor de x es 0, seguimos la línea punteada que sale del nodo x y llegamos al nodo izquierdo y. Dado que el valor de y es 1, seguimos la línea sólida que sale del nodo y y llegamos al nodo terminal que se encuentra hasta la izquierda del árbol, el cual tiene como valor 0. Por lo tanto podemos concluir que $f(0,1) = 0$.

Si hacemos lo mismo para $f(0,0)$ llegaremos finalmente hacia el nodo terminal cuyo valor es 1. Se puede observar que las otras dos posibilidades llegan a nodos terminales con valor 0. De lo cual podemos finalmente decir que el árbol binario de decisión de la Figura 2.7 representa la función booleana $f(x,y) = \overline{x + y}$.

En lo que a tamaño concierte, los árboles binarios de decisión son muy parecidos a las tablas de verdad. Si la función booleana f depende de n variables booleanas, entonces el árbol binario de decisión que representa dicha función tendrá al menos $2^{n+1} - 1$ nodos. Siendo que la tabla de verdad para la misma función tendrá 2^n líneas, resulta evidente que los árboles de decisión no son una representación mucho más compacta para funciones booleanas. Sin embargo, los árboles binarios de decisión contienen cierta redundancia que puede ser explotada para producir estructuras de menor tamaño.

Dado que los nodos terminales de un árbol binario de decisión pueden contener únicamente 0 o 1 como valor, podemos optimizar la representación colocando apuntadores a una única copia de 0 y una única copia de 1.

Por ejemplo, el árbol binario de decisión de la Figura 2.7 puede ser optimizado de esta manera y producir la estructura que se muestra a continuación:

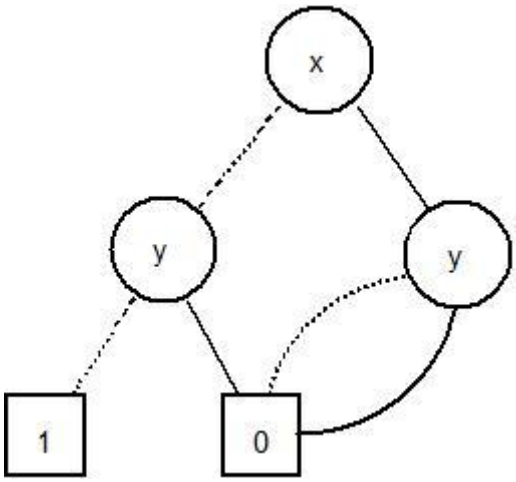


Figura 2.8: Árbol binario de decisión con nodos terminales únicos

Aunque con esta optimización se ahorra cierto espacio, se conserva el número de transiciones representadas.

Una segunda optimización al árbol de la Figura 2.8 podría ser eliminar los puntos de decisión innecesarios. Por ejemplo, el nodo y derecho es innecesario dado que ambas transiciones que salen de él nos llevan al mismo nodo terminal, por lo tanto podemos eliminarlo y obtener la siguiente estructura:

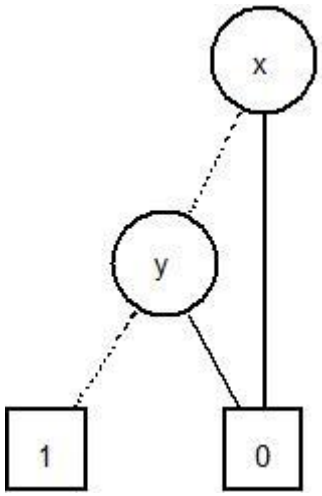


Figura 2.9: Árbol binario de decisión optimizado

La Figura 2.8 y 2.9 en realidad ya son ejemplos de diagramas binarios de decisión (*BDDs*). Como puede observarse son estructuras mucho más generales que los árboles binarios de decisión.

Existe una tercera optimización que puede realizarse a los árboles binarios de decisión, en esta optimización lo que se busca son *sub-BDDs* repetidos. Un *sub-BDD* es una parte del *BDD* que ocurre por debajo de un nodo cualquiera.

Un ejemplo de lo anterior puede observarse en la siguiente figura:

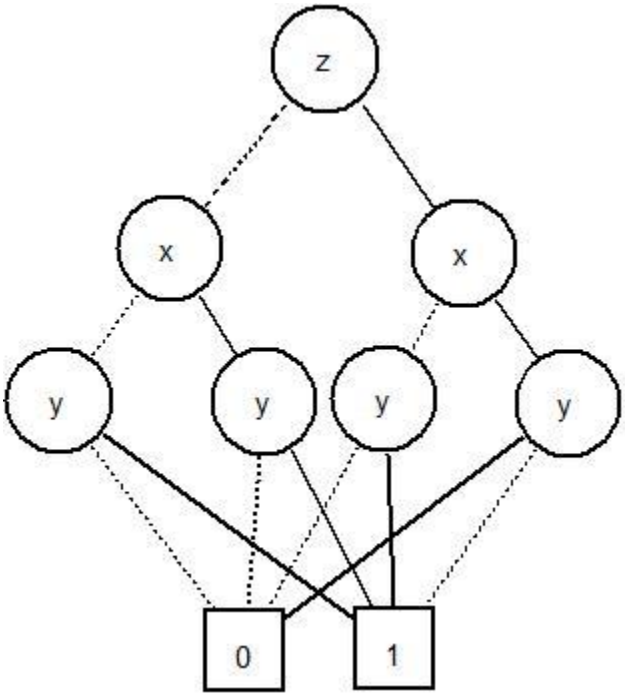


Figura 2.10: BDD con un sub-BDD duplicado

Por ejemplo, los dos nodos interiores y llevan a cabo la misma acción, esto debido a que el sub-*BDD* debajo de ellos tiene la misma estructura. Por lo tanto, uno de los dos nodos interiores y puede ser removido del *BDD*, resultando en el siguiente *BDD*:

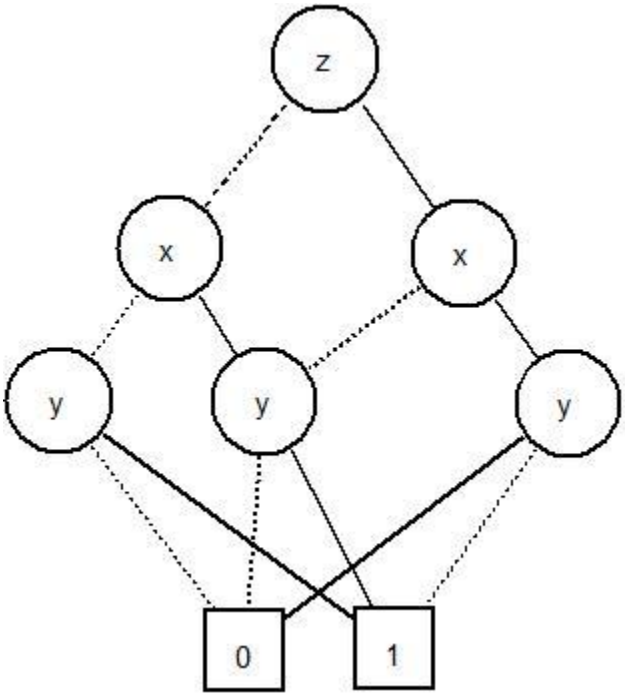


Figura 2.11: BDD con eliminación de nodo repetido

Finalmente, podemos observar que el nodo y izquierdo y el nodo y central también pueden ser combinados en uno solo pues el *sub-BDD* debajo de ellos tiene la misma estructura. Al hacer esta eliminación resulta en que ambas salidas del nodo x izquierdo van al mismo lugar por lo tanto podemos eliminar ese punto de decisión como lo hicimos antes en la Figura 2.9, finalmente obtenemos el *BDD* reducido de la Figura 2.12.

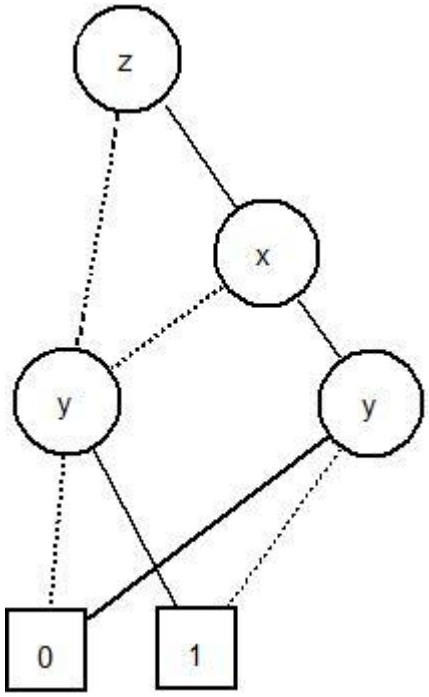


Figura 2.12: BDD reducido

En resumen, existen tres diferentes acciones que podemos llevar a cabo para reducir un *BDD*:

C1. Eliminar nodos terminales duplicados.

C2. Eliminar puntos de decisión redundantes.

C3. Eliminar nodos no terminales duplicados.

Se dice que un *BDD* es un *BDD* reducido si ninguna de las acciones C1 al C3 puede ser aplicada sobre el *BDD*, es decir, no se puede hacer ninguna otra reducción.

Mediante las tres reducciones anteriores podemos obtener estructuras de datos que nos permiten representar funciones booleanas de forma compacta. Sin embargo, existe otra situación que debemos considerar al construir *BDDs*. Los *BDDs* con múltiples ocurrencias de una variable booleana a través de una trayectoria resultan bastante ineficientes. Además, parece no haber una forma simple para decidir si dos *BDDs* son equivalentes, es decir, puede darse el caso de que dos *BDDs* representen la misma función booleana pero su estructura no sea la misma aún cuando ambos sean reducidos.

Esto puede mejorarse si se impone un orden en la ocurrencia de variables a lo largo de las trayectorias del diagrama. En general, la elección de un orden hace una gran diferencia en el

tamaño del *BDD* resultante. A este tipo de *BDD* se les llama Diagramas Binarios de Decisión Ordenados (*OBDD* por sus siglas en inglés).

Tomemos en cuenta la siguiente definición: Sean B_1 y B_2 dos *OBDD*. Se dice que su ordenamiento es compatible si no existen variables x y y tal que x aparezca antes que y en el ordenamiento de B_1 y y aparezca antes que x en el ordenamiento de B_2.

Tenemos el siguiente teorema: El *OBDD* reducido que representa una función f es único. Es decir, sea B y B' dos *OBDD* reducidos con un ordenamiento compatible. Si B y B' representan la misma función booleana, entonces ambos deben tener la misma estructura.

Por lo tanto, dados dos *OBDD* reducidos que representen la misma función booleana, y dado que sus ordenamientos sean iguales, resulta trivial la determinación de equivalencia entre ambos, ya que ésta se reduce a revisar si ambos *OBDD* tienen la misma estructura. De esta manera podemos concluir que los *OBDD* tienen una forma canónica, es decir un único *OBDD* reducido.

2.8.2 Algoritmos para OBDD reducidos

Aquí se describe de forma general los principales algoritmos asociados al uso de *OBDDs* reducidos. Estos algoritmos sirven para llevar a cabo las operaciones básicas que permiten su implementación en sistemas como el que se desarrollo en este trabajo de tesis.

No se pretende entrar en los detalles de implementación de cada uno de los algoritmos pues estos no son el objetivo del trabajo; las descripciones que se dan pueden encontrarse con mucho más detalle en [4].

Los *OBDDs* que pretendemos usar deben ser reducidos de tal manera que obtengamos estructuras canónicas de representación, además de que aseguremos el mayor ahorro de espacio posible e cuanto a tamaño de la estructura. El algoritmo $reduce$ lleva a cabo la reducción del *OBDD* mediante los pasos de optimización C1, C2 y C3. Este algoritmo recorre el *OBDD* etiqueta por etiqueta de abajo hacia arriba, comenzando por los nodos terminales. Durante el recorrido del *OBDD*, el algoritmo asigna etiquetas $id(n)$ con valores enteros a cada nodo n en el *OBDD*, de tal manera que los sub-BDD con nodos raíz n y m denotan la mima función booleana, si, y solo si, $id(n)$ es igual que $id(m)$. Resulta que los tres pasos de optimización que vimos antes se

resuelven mediante la comparación de etiquetas, recordando que la asignación de etiquetas se hace de abajo hacia arriba comenzando por asignar las mismas etiquetas a nodos terminales con iguales valores.

Otro algoritmo esencial para la manipulación de *OBDD* reducidos es el algoritmo $apply$. El algoritmo $apply$ se utiliza para implementar operaciones booleanas como pueden ser la conjunción, la disyunción, el complemento, etc.

El algoritmo $apply$ opera de forma recursiva sobre dos estructuras *OBDD* B_f y B_g, de la siguiente forma:

1. Sea v la variable hasta la izquierda en la lista ordenada de variables que ocurre tanto en B_f como en B_g, es decir, se comienza por las raíces y se avanza por la lista hacia la derecha.
2. Dividir el problema en dos sub-problemas con v siendo 0 y v siendo 1 y resolverlo recursivamente.
3. En las hojas, aplicar la operación booleana directamente.

Una llamada al algoritmo $apply$ se vería de la siguiente forma: $apply(op, B_f, B_g)$ en donde op sería el operador booleano que se desea aplicar sobre ambos *OBDD*.

Resulta entonces en un algoritmo que va recorriendo ambas estructuras de arriba hacia abajo. Conforme el algoritmo va bajando por las ramas del *OBDD* se aplica (desde la raíz) la expansión de Shannon para todas las variables. La expansión de Shannon se define de la siguiente manera:

Para todas las fórmulas booleanas f y todas las variables booleanas x (incluso aquellas que no ocurren en f) tenemos

$$f \equiv \bar{x} \cdot f[0/x] + x \cdot f[1/x]$$

La función $apply$ se basa en la expansión de Shannon para $f\ op\ g$:

$$f\ op\ g = \bar{x}_i \cdot (f[0/x_i] op\ g[0/x_i]) + x_i \cdot (f[1/x_i] op\ g[1/x_i])$$

En donde $f[0/x]$ denota la función booleana que se obtiene al reemplazar todas las ocurrencias de x en f por 0. $f[1/x]$ se define de forma similar. Ambas expresiones son llamas restricciones de f.

El resultado del algoritmo $apply$ es un *OBDD*, pero no necesariamente reducido. Las reducciones pueden irse aplicando conforme se ejecuta $apply$ o al final.

Otro algoritmo importante para la manipulación de *OBDDs* es el algoritmo $restrict$. Dado un *OBDD* B_f el cual representa una fórmula booleana f, empleamos el algoritmo $restrict$ de la siguiente manera $restrict(0, x, B_f)$ para obtener el *OBDD* reducido que representa la restricción $f[0/x]$ usando el mismo ordenamiento de variables que B_f.

Finalmente, damos el algoritmo $exists$. Este algoritmo se puede ver como una forma de "relajar" una función booleana. Formalmente $exists$ se define de la siguiente forma:

$$\exists x.f \stackrel{\text{def}}{=} f[0/x] + f[1/x]$$

Es decir, buscamos una x que pueda hacer verdadera a f al ponerle valores 0 o 1 a x.

El algoritmo $exists$ puede ser expresado en términos de los algoritmos antes vistos de la siguiente forma:

$$apply(+, restrict(0, x, B_f), restrict(1, x, B_f))$$

2.8.3 Verificación de modelos con diagramas binarios de decisión

El uso de *BDDs* en la verificación de modelos resultó en un cambio radical en la década de 1990 debido a que estas estructuras de datos permitieron verificar sistemas con bastantes más estados que lo que se había podido hacer antes.

El pseudo-código que se presentó antes para el algoritmo de etiquetamiento consiste en manipular conjuntos de estados. La idea al utilizar *BDDs* es almacenar en ellos el modelo y los conjuntos de estados.

Aquí se presentan las ideas utilizadas para llevar a cabo la implementación del algoritmo de etiquetamiento utilizando *BDDs*.

Una de las partes principales al especificar el modelo M a verificar es la función de transición que une los distintos estados del sistema. La función de transición es un subconjunto de SxS en donde S es el conjunto de estados en el sistema como lo hemos estado usando hasta ahora. Estos conjuntos de estados pueden ser representados como *OBDDs*.

Para especificar la forma en que se deberá definir el *OBDD* que representa la función de transición utilizaremos el siguiente modelo:

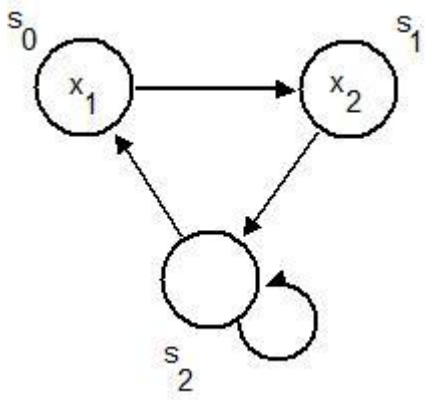


Figura 2.13: Modelo simple

Para representar la función de transición de la Figura 2.13 podemos construir la siguiente tabla de verdad:

x_1	x_2	x_1'	x_2'	$\rightarrow$
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0

1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Tabla 1. Función de transición de la Figura 2.13

En la Tabla 1, las variables x_1' y x_2' representan el valor de la variable x_1 y x_2 respectivamente después de la transición. Por lo tanto, los vectores de la tabla se construyen de la forma $((v_1, v_2, \dots, v_n), (v_1', v_2', \dots, v_n'))$, en donde v_i es 1 si $p_i \in L(s)$ es decir si la variable en el estado se encuentra en la función de etiquetamiento, y 0 en otro caso.

Por lo tanto, para construir el *OBDD* que representa esta transición, debemos representar la función booleana:

$$(l_1 \cdot l_2 \cdots l_n) \cdot (l_1' \cdot l_2' \cdots l_n')$$

Y entonces la definición de todas las transiciones del modelo, i.e. todas las transiciones en la tabla, se representan mediante la disyunción de fórmulas como la anterior. Para construir dicho *OBDD* basta considerar las transiciones que se encuentren contenidas dentro de la función de transición definida en el modelo, i.e. solo los renglones en los cuales $\rightarrow$ tenga valor 1 en la tabla.

La función booleana para el ejemplo en la Figura 2.13 quedaría definida como es muestra a continuación:

$$f^{\rightarrow} \stackrel{\text{def}}{\equiv} \overline{x_1} \cdot \overline{x_2} \cdot \overline{x'_1} \cdot \overline{x'_2} + \overline{x_1} \cdot \overline{x_2} \cdot x'_1 \cdot \overline{x'_2} + x_1 \cdot \overline{x_2} \cdot \overline{x'_1} \cdot x'_2 + \overline{x_1} \cdot x_2 \cdot \overline{x'_1} \cdot \overline{x'_2}$$

El *OBDD* resultante es el siguiente:

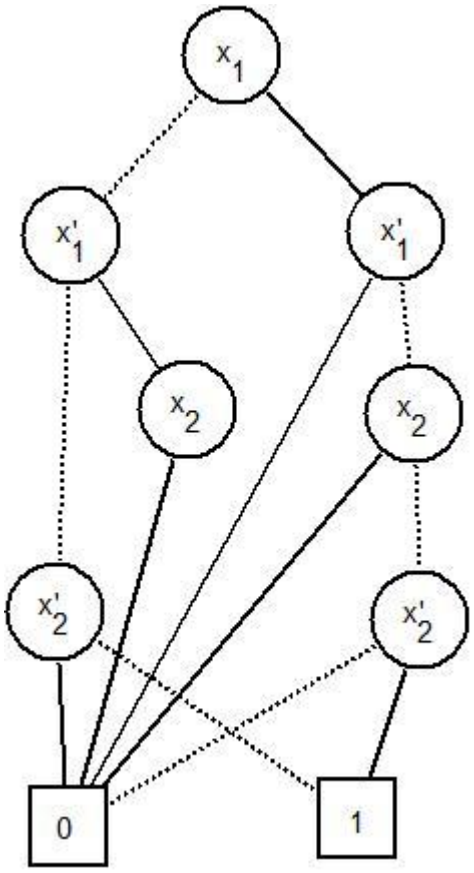


Figura 2.14: OBDD para la función de transición del modelo en la Figura 2.13

Una vez que sabemos cómo representar el modelo a verificar, necesitamos definir ciertos cambios en el pseudo-código del algoritmo de etiquetamiento.

Los principales cambios se llevan a cabo en las dos primitivas $pre_{\exists}(X)$ y $pre_{\forall}(X)$. Estas primitivas pueden ser obtenidas, dados los *OBDDs* B_x para X y $B_{\rightarrow}$ para la función de transición $\rightarrow$.

Como vimos antes, $pre_{\forall}$ puede ser definido en términos de $pre_{\exists}$, por lo tanto basta con definir uno de los dos.

El procedimiento sugerido para obtener $pre_{\exists}$ se define en los siguientes pasos:

1. Renombrar las variables en B_x a sus versiones primas, i.e. las variables después de la transición; al *OBDD* resultante le llamamos $B_{x'}$.
2. Obtener el *OBDD* para $exists(\vec{x'}, apply(\cdot, B_{\rightarrow}, B_{X'}))$.

Es decir, buscamos estados que tengan transición con los estados primos en $B_{X'}$. El *OBDD* resultante será lo que regrese la primitiva $pre_{\exists}$.

Además, cuando se especificó el pseudo-código para el algoritmo de etiquetamiento, se definió el uso de operaciones booleanas. Asimismo, se especificó que existe un acceso total al modelo desde cualquier parte del algoritmo. Resulta evidente que cualquier uso de operaciones booleanas deberá ser sustituido por llamadas a operaciones booleanas mediante el algoritmo $apply$.

CAPÍTULO 3
REDES GENÉTICAS

3. 1 Introducción

Basamos la discusión sobre formalismos para representar redes genéticas principalmente en [10], mientras que la sección de casos de estudio principalmente en [12] y [13].

Enfocamos este trabajo principalmente al desarrollo de una herramienta computacional que permita la verificación de cierto tipo de modelos. Como veremos más adelante, el caso de estudio y por lo tanto el modelo objetivo a verificar son redes genéticas. De aquí la importancia de entenderlas hasta cierto grado, sin perder de vista el objetivo principal: la construcción y conclusión de un verificador de modelos simbólico que sirva como apoyo para el estudio de fenómenos biológicos.

Este capítulo trata de forma general el tema de redes genéticas, cubriendo primero una explicación general y posteriormente dedicando un espacio a la definición y explicación de diferentes modelos utilizados comúnmente en la biología para representarlas y estudiarlas. Después presentamos una discusión sobre cómo y por qué elegir un formalismo de entre todos los disponibles; en particular defendemos el uso de redes booleanas. Finalmente describimos los casos de estudio que nos sirvieron como línea principal para las pruebas del verificador.

3. 1. 1 Sobre las redes genéticas

El genoma de un organismo juega un rol central en el control de los procesos celulares, como por ejemplo la respuesta de una célula a señales ambientales, o la replicación del ácido desoxirribonucleico (*ADN*) que precede a la división de celular.

Las proteínas que sintetizan los genes pueden funcionar como factores que ligan ciertos mecanismos de regulación con otros genes, por ejemplo como enzimas que catalizan reacciones metabólicas, o como componentes que funcionan como caminos para transformar señales de un tipo a otro.

En general todas las células en un organismo contienen el mismo material genético; con excepción de las células sexuales o gametos. Esto implica que para poder entender cómo los genes son involucrados en los procesos de control intracelular y los procesos intercelulares, el alcance de estudio debe ser ampliado desde secuencias de nucleótidos hasta sistemas de regulación en los cuales se determina qué genes se deben expresar, cuándo y dónde en el organismo, y con qué alcance.

La expresión de genes es un proceso complejo regulado en varias etapas durante la síntesis de proteínas. La expresión de un gen puede ser controlada durante el procesamiento, transporte y traducción del ácido ribonucleico (*ARN*). La degradación de proteínas y de productos intermedios de *ARN* también puede ser regulada al interior de la célula. Las proteínas que satisfacen las funciones de regulación antes mencionadas son también producidas por otros genes. Todo esto da paso al nacimiento de los sistemas genéticos de regulación los cuales son estructurados mediante redes de interacciones de regulación entre el *ADN*, *ARN*, proteínas y pequeñas moléculas.

En la siguiente figura se muestra un ejemplo de una red genética de regulación. Se trata de una red simple, en la cual se encuentran involucrados tres genes que codifican proteínas que inhiben la expresión de otros genes.

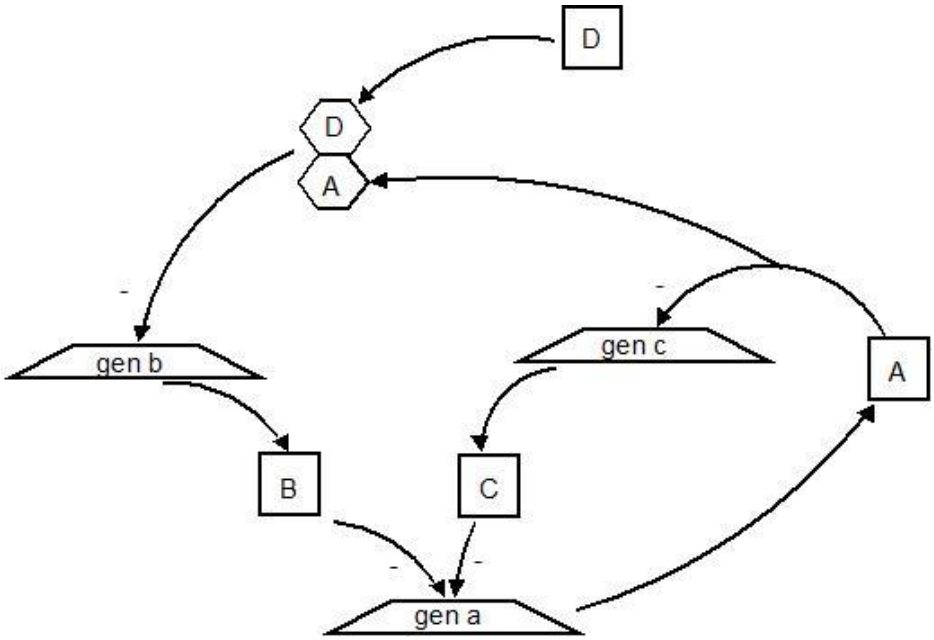


Figura 3.1: Ejemplo de una red genética de regulación simple

En la figura anterior las letras A, B, C y D representan proteínas represoras. Las flechas con signos negativos son regulaciones negativas en la red, mientras que las flechas sin signo significan una transformación en la proteína. En la red ejemplo, las proteínas B y C de forma independiente inhiben la expresión del gen a, mientras que las proteínas A y D interactúan para inhibir conjuntamente la expresión del gen b. La interacción que se lleva a cabo entre los diferentes genes es de gran interés para la biología; las redes como la presentada en la Figura 3.1 se modelan utilizando ciertas técnicas que se discutirán más adelante.

Durante años se han analizado grandes cantidades de datos que han permitido y contribuido al descubrimiento de un gran número de genes y sus formas de regulación. La base de datos Kyoto Encyclopedia of Genes and Genomes (*KEGG*), por ejemplo, contiene información sobre alrededor de 4,250,322 genes de 912 especies [M2]. En muchos casos, las proteínas involucradas en el control de la expresión de estos genes, así como los mecanismos moleculares a través de los cuales se lleva a cabo la regulación genética han sido identificados. El entendimiento de los patrones y comportamientos que resultan de la interacción entre genes en redes genéticas significa un gran reto científico con un gran potencial industrial.

El desarrollo reciente de técnicas experimentales ha permitido incrementar de forma considerable la velocidad e interés en el estudio de las redes genéticas. Algunas de estas técnicas han permitido

caracterizar el estado de las células hasta niveles antes no imaginables; se han convertido en herramientas esenciales para experimentar y obtener información sobre la forma en que se expresan los genes.

En paralelo con las herramientas experimentales que se han desarrollado en biología, los métodos formales para llevar a cabo el modelado y simulación de los procesos de regulación genética son indispensables. La mayoría de las redes genéticas de interés involucran grandes cantidades de genes conectados entre sí, por lo que un entendimiento intuitivo sin el uso de herramientas computacionales es prácticamente impensable.

Si se hace uso de métodos formales para el estudio de redes genéticas, entonces se pueden generar predicciones de sus comportamientos de forma sistemática y por lo tanto validable. Además, las herramientas computacionales permiten investigar redes genéticas mucho más complejas y grandes.

En la Figura 3.2 se muestra del lado izquierdo el ciclo básico mediante el cual se modelan y simulan las redes genéticas con uso de herramientas computacionales y del lado derecho el ciclo mediante el cual se obtienen los resultados por medio de herramientas experimentales:

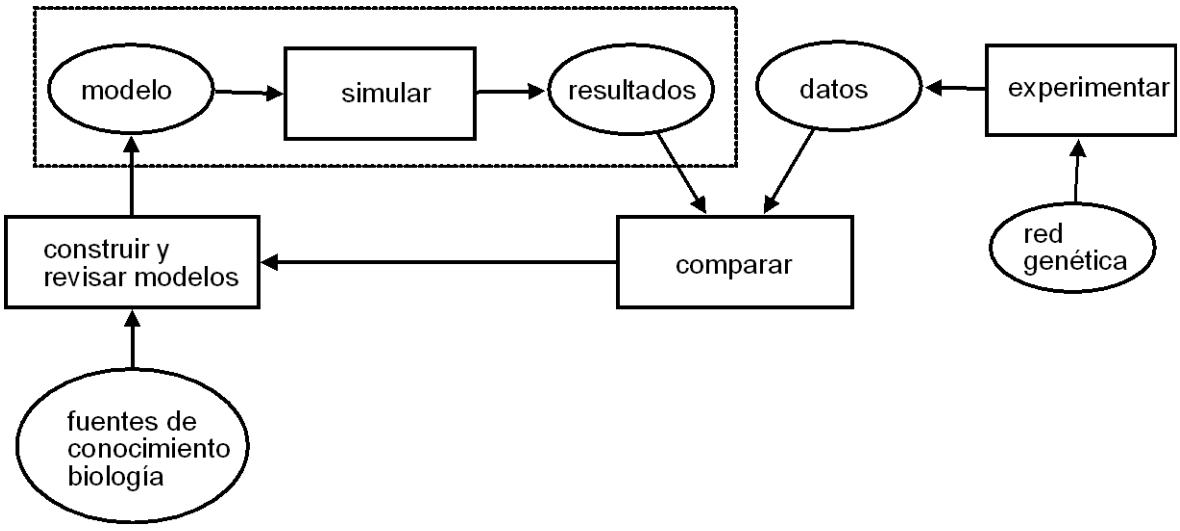


Figura 3.2: Ciclo de modelado-simulación. Las cajas representan actividades, los óvalos fuentes de información, las flechas son flujos de información. La caja punteada representa el alcance del verificador de modelos.

El ciclo básico para modelar/simular sistemas de regulación como lo son las redes genéticas comienza con la creación de un modelo inicial, el cual se construye obteniendo información de diversas fuentes de conocimiento. Una vez que se tiene el modelo se simula mediante el uso de herramientas computacionales. En el caso de este trabajo, deseamos sustituir la simulación por las técnicas desarrolladas en el área de verificación de modelos. El cuadro punteado en la Figura 3.2 representa a manera de caja negra los principales bloques que componen al verificador de modelos desarrollado en este trabajo de tesis, en donde tenemos un modelo como entrada, los procesos de verificación de modelos que a partir de las reglas de la red verifican propiedades y finalmente tenemos como salida una serie de resultados.

La acción de comparar los resultados/predicciones de la red genética con los datos obtenidos de la experimentación sirve como un excelente indicador sobre la validez del modelo. En general, si se considera que los datos obtenidos de la experimentación son confiables, se espera que el modelo deba ser revisado y en su caso ajustado. En la Figura 3.2 se observa un ciclo que va de la comparación a la construcción del modelo. Dicho ciclo se puede y debe repetir tantas veces sea necesario hasta obtener el modelo adecuado. Este es uno de los procesos en el cual el verificador de modelos desarrollado es de enorme utilidad.

Las bases formales para la herramienta computacional que desarrollamos radican en métodos que se han estudiado en las áreas de la biología matemática y la bioinformática. Desde la década de 1960, varios formalismos matemáticos se desarrollaron para describir redes de regulación como las redes genéticas. Estos formalismos se complementan con técnicas de simulación (en nuestro caso sustituimos la simulación con la verificación de modelos) para producir predicciones a partir de un modelo del sistema de interés, y con técnicas para construir modelos a partir de datos experimentales y bases de conocimiento.

Tradicionalmente el énfasis se ha centrado en las técnicas de simulación, y se asume que los modelos se construyen “a mano” utilizando datos experimentales que se encuentran en la literatura.

En las siguientes secciones se discuten algunos de los formalismos matemáticos que existen, entre ellos el que utiliza el grupo de biólogos con los que se trabajó durante el desarrollo del verificador.

3. 2 Representación de redes genéticas

Los formalismos matemáticos que se describen en esta sección incluyen el uso de gráficas dirigidas, redes bayesianas, redes booleanas y una generalización de redes booleanas.

3.2.1 Gráficas dirigidas

Una de las formas más directas para modelar redes genéticas es usando gráficas dirigidas. Una gráfica dirigida G se define como un par $\langle V, E \rangle$, en donde V es el conjunto de vértices y E el conjunto de aristas. Una arista dirigida es un par (i, j) de vértices, en donde a i se le llama el origen y a j el destino. Los vértices de una gráfica dirigida corresponden a los genes del sistema que se modela, mientras que las aristas denotan interacciones entre los genes del sistema.

Las gráficas que se utilizan para representar gráficas dirigidas se pueden generalizar de varias formas. Los vértices y las aristas se pueden etiquetar para permitir representar información adicional sobre los genes y su interacción.

De lo anterior, se puede definir una arista dirigida como una tupla (i, j, s) en donde s es $+$ o $-$, de tal forma que se pueda indicar si el gen j es activado o inhibido por el gen i.

En la siguiente figura se puede observar un ejemplo de gráfica dirigida con las características antes mencionadas:

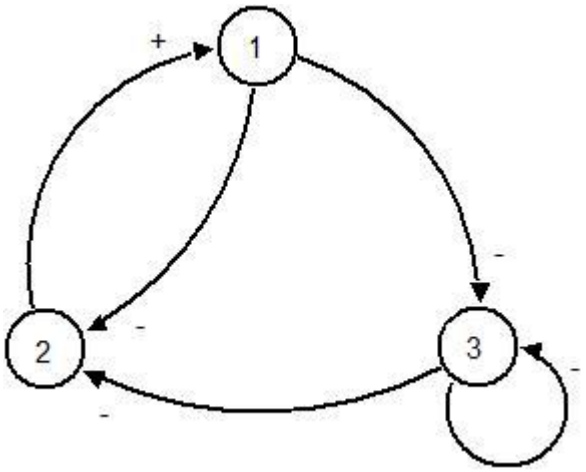


Figura 3.3: Representación de red genética

La gráfica de la Figura 3.3 en términos de la especificación dada anteriormente quedaría definida como se muestra a continuación:

$$V = \{1, 2, 3\}$$

$$E = \{(1, 2, -), (1, 3, -), (2, 1, +), (3, 2, -), (3, 3, -)\}$$

Existen varias bases de datos con información suficiente para la construcción de este tipo de gráficas que permite representar la interacción entre genes. Algunos ejemplos de estas bases de conocimiento son:

- aMAZE
- EcoCyc
- GeneNet
- GeNet
- KEGG
- KNIFE
- RegulonDB

Por ejemplo, en la base de datos GeneNet se describen diversos tipos de células, genes con sus características de regulación, proteínas, interacciones regulatorias y reacciones químicas. Las bases de conocimiento existentes regularmente incluyen también aplicaciones que permiten editar las redes incluso hasta el nivel de poder manipular interacciones de forma individual. Otra característica interesante en este tipo de aplicaciones es la posibilidad de visualizar las redes genéticas a diferentes niveles de detalle.

Se pueden realizar varias operaciones sobre gráficas dirigidas de tal manera que se puedan llevar a cabo las predicciones necesarias sobre las redes genéticas que se estén estudiando.

Por ejemplo, la búsqueda de trayectorias entre dos genes puede revelar la falta de interacciones regulatorias o proveer de indicios sobre cierta redundancia existente en la red modelada. Los ciclos en la gráfica pueden indicar relaciones de retroalimentación que son de interés para ciertos procesos en la biología. La conectividad global de una red puede ser un indicador de la complejidad y la forma en que los genes son regulados. Una conectividad baja en subgráficas se

puede observar biológicamente como submódulos con comportamientos aislados dentro del sistema de regulación.

El uso de gráficas como representación de redes genéticas resulta en un formalismo matemático bastante directo, fácil de entender y utilizar ya que las gráficas son objetos matemáticos ampliamente utilizados y conocidos.

3.2.2 Redes bayesianas

En el formalismo de redes bayesianas, la estructura de una red genética se modela mediante una gráfica acíclica dirigida (*DAG* por sus siglas en inglés), la cual se define como $G = \langle V, E \rangle$. Los vértices $i \in V$ en donde $1 \leq i \leq n$, representan genes y corresponden a variables aleatorias X_i. Si i es un gen, entonces X_i describirá el nivel de expresión de i. Por cada X_i, una distribución condicional del tipo $p(X_i \mid parents(X_i))$ se debe definir, en donde $parents(X_i)$ denota a las variables que representan a los reguladores directos de i en G. La gráfica G y las distribuciones condicionales $p(X_i \mid parents(X_i))$ en conjunto especifican la probabilidad de distribución conjunta $p(\boldsymbol{X})$ para la red bayesiana.

Para redes bayesianas se puede definir la independencia condicional de la siguiente forma $i(X_i; \boldsymbol{Y} \mid \boldsymbol{Z})$ lo cual expresa el hecho de que X_i es independiente de $\boldsymbol{Y}$ dado $\boldsymbol{Z}$, en donde $\boldsymbol{Y}$ y $\boldsymbol{Z}$ son conjuntos de variables. En este tipo de representación se asume que los estados en el modelo dependen solo de sus antecesores directos $(parents)$, de tal manera que por cada gen i en G, tenemos que la independencia condicional se define como $i(X_i; nondescendants(X_i)|parents(X_i))$. La distribución de probabilidad conjunta se define como:

$$p(\boldsymbol{X}) = \prod_{i=1}^{n} p(X_i|parents(X_i))$$

La siguiente figura muestra un ejemplo de este tipo de representación:

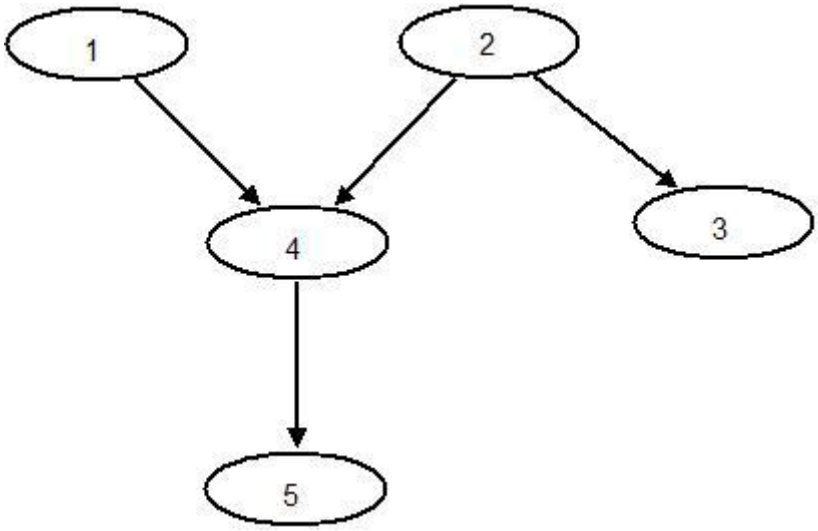


Figura 3.4: Red bayesiana como representación de red genética

Las distribuciones de probabilidad para las variables aleatorias quedarían definidas como:

$$p(X_1), p(X_2), p(X_4|X_1, X_2), p(X_5|X_4), p(X_3|X_2)$$

La distribución de probabilidad conjunta:

$$p(\boldsymbol{X}) = p(X_5|X_4)p(X_4|X_1, X_2)p(X_3|X_2)p(X_2)p(X_1)$$

Independencias condicionales:

$$i(X_1; X_2, X_3), i(X_2; X_1), i(X_4; X_3|X_1, X_2)$$

$$i(X_3; X_1, X_4, X_5|X_2), i(X_5; X_1, X_2, X_3|X_4)$$

La equivalencia entre redes bayesianas se puede identificar mediante el conjunto de independencias, es decir, si dos redes bayesianas implican el mismo conjunto de independencias condicionales, se dice que ambas redes bayesianas son equivalentes.

El enfoque mediante redes bayesianas para modelar redes de regulación como las redes genéticas es atractivo ya que tiene bases estadísticas sólidas, lo cual le permite al modelo lidiar con aspectos

estocásticos sobre la expresión de los genes y resolver de forma casi natural problemas como el ruido en la toma de medidas experimentales.

Otra ventaja de las redes bayesianas, es que estas pueden ser utilizadas aún cuando exista información incompleta sobre el sistema que se desea modelar.

Las redes bayesianas son representaciones intuitivas para las redes genéticas. Sin embargo, tienen cuando menos una gran desventaja, recordando de las gráficas dirigidas en el punto anterior. La posibilidad de representar ciclos y por lo tanto de poder detectarlos en el modelo es de gran valor para la biología ya que implica ciertas propiedades sobre el sistema. No obstante, las redes bayesianas como tal no pueden modelar este tipo de propiedades ya que como se mencionó antes, una red bayesiana es una gráfica dirigida acíclica.

3.2.3 Redes booleanas

Las redes booleanas resultan de mayor interés para este trabajo pues es el formalismo que utilizan actualmente el grupo de biólogos con el cual trabajamos. Uno de los parámetros de entrada que recibe el verificador de modelos desarrollado, es definido a partir de la representación como red booleana de una red genética.

El modelado de redes genéticas mediante el formalismo de redes booleanas cobró popularidad tras los estudios publicados por Kauffman. En este tipo de representación, el estado de la expresión de un gen dentro de una red genética se describe mediante una variable booleana. El valor de dicha variable indica si el gen se encuentra activo (variable en 1, verdadero), o si el gen se encuentra inactivo (variable en 0, falso). La interacción entre elementos de la red se representa mediante funciones booleanas que sirven para obtener el estado del gen dado el valor presente en genes relacionados. Al combinar la representación de genes mediante variables booleanas y utilizar funciones booleanas para la interacción entre genes se obtiene una red booleana.

Para continuar con la definición de red booleana, suponga el vector $\hat{x}$ de variables el cual representa el estado de una red genética con n elementos, en donde cada x_i puede tener valor 1 o 0, de tal forma que el sistema consiste de 2^n estados. El estado x_i de un elemento en el tiempo $t+1$ se obtiene mediante una función booleana dados k de los n estados en el tiempo t (k puede

ser diferente por cada x_i. La variable x_i puede ser vista como salida y las k variables como la entrada de la función booleana siendo evaluada.

En resumen, la dinámica de una red booleana que describe una red genética se puede expresar como:

$$x_i(t+1) = f_i\left(\hat{\boldsymbol{x}}(t)\right), 1 \leq i \leq n$$

En donde f_i es la función que tiene como salida la variable x_i y como entrada k variables en el tiempo inmediato anterior.

En la siguiente figura se muestra una red booleana dibujada en forma de diagrama en el cual se expresa claramente el estado de las variables en el tiempo t y el consecuente estado en el tiempo $t+1$. La función booleana que mapea de un estado a otro a partir de las variables involucradas se especifica debajo de cada elemento:

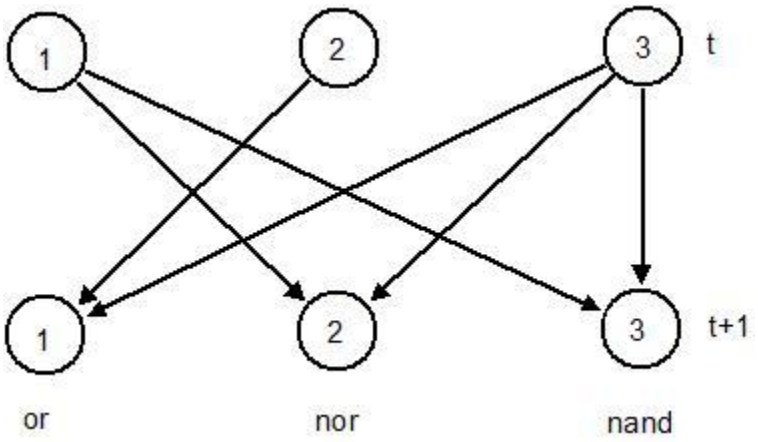


Figura 3.5: Diagrama de una red booleana

La Figura 3.5 es una representación sencilla para observar la transición entre estados. La transición de un estado al siguiente se determina de forma paralela, aplicando la función booleana por cada elemento que tenga como entrada. Por ejemplo, dado un vector de estado 000 en el tiempo $t = 0$, el sistema en la Figura 3.5 se moverá hacia el estado 011 para el tiempo $t = 1$. Es decir, en términos de redes genéticas, si todos los genes se encuentran inactivos en el tiempo $t = 0$, entonces dada la dinámica de la red, resultará que en el tiempo $t = 1$, los genes 2 y 3 estarán

activos. Bajo este esquema, las redes booleanas son en principio determinísticas en el sentido de que dada una entrada habrá una única salida, y además, sus funciones de transición son síncronas, en el sentido de que todas las salidas de los genes se actualizan al mismo tiempo.

Existen otras formas de representar redes booleanas. La siguiente figura muestra la misma red de la Figura 3.5 como diagrama de bloques lógicos:

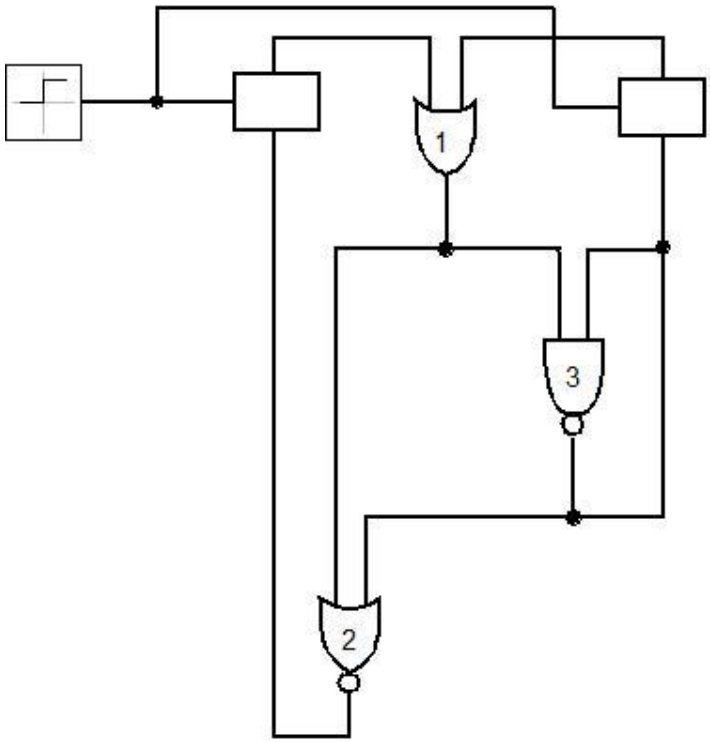


Figura 3.6: Red booleana como diagrama de bloques lógicos

Las ecuaciones que especifican tanto la Figura 3.5 como la 3.6 son las siguientes:

$$x_1(t+1) = x_2(t)\ or\ x_3(t)$$

$$x_2(t+1) = x_1(t)\ nor\ x_3(t)$$

$$x_3(t+1) = x_1(t)\ nand\ x_3(t)$$

A partir de ellas también es fácil de obtener el estado de la red para cualquier tiempo t. Como puede verse, existen diversas maneras para representar redes booleanas. En particular, nuestro verificador de modelos utiliza una representación en forma de tablas de verdad; esto se verá con más detalle en capítulos posteriores.

En las redes booleanas, una secuencia de estados conectados mediante transiciones forman una *trayectoria* en el sistema. El número de estados en el espacio de estados es finito, pero el número de estados en una trayectoria es infinito. Todos los estados iniciales de una trayectoria alcanzarán tarde o temprano un estado estable o un estado que forme un ciclo dentro de la red, a estos estados se les llama *atractores*. Los estados que no forman parte de un *atractor* se llaman estados *transitorios*. Los estados atractores y los estados transitorios que llevan hacia el atractor en conjunto conforman la *cuenca de atracción*.

Para redes simples, los atractores y sus cuencas de atracción pueden ser calculados a mano, pero para sistemas grandes el uso de herramientas computacionales es indispensable. Se han desarrollado herramientas para apoyar el estudio de este tipo de estados; en nuestro caso la propuesta es utilizar *CTL* con las propiedades mencionadas en el capítulo anterior.

Las redes booleanas fueron de los primeros formalismos para los cuales se propusieron métodos que permitieran inducir propiedades sobre los sistemas siendo modelados. Un ejemplo de esto es el algoritmo *REVEAL* desarrollado por Liang en 1998. En general, el algoritmo *REVEAL* hace uso de la teoría de la información para establecer cómo los elementos dentro de una red se encuentran conectados y después determinar las funciones que especifican la lógica de interacción.

Las redes booleanas permiten especificar redes genéticas de gran tamaño de forma simple y eficiente debido a ciertas suposiciones que simplifican la definición del estado de los genes y la definición de las interacciones entre genes. En el formalismo de redes booleanas, un gen es considerado encendido o apagado (activo o inactivo) y cualquier otro estado es omitido de la especificación. Otra simplificación en este tipo de redes, es la suposición de que las transiciones entre los diversos estados de activación que componen la red se llevan a cabo de forma síncrona.

3.2.4 Redes lógicas generalizadas

Las redes lógicas generalizadas se basan en la generalización de redes booleanas en el sentido de que permiten a las variables del sistema tener más de dos valores y además suponen que las transiciones entre estados pueden ocurrir de forma asíncrona.

Esta generalización desarrollada por Thomas en la década de 1970, utiliza variables discretas x_i, llamadas *variables lógicas*. Como se verá a continuación, los posibles valores de x_i son definidos al comparar conjuntos de valores x_i con un cierto umbral de influencia de i en otros elementos del

sistema de regulación. Si un elemento i tiene influencia sobre otros p elementos del sistema de regulación, entonces i podrá tener tantos como p umbrales:

$$\alpha_i^{(1)} < \alpha_i^{(2)} < \cdots < \alpha_i^{(p)}$$

Dados los umbrales $\alpha_i^{(1)}$ hasta $\alpha_i^{(p)}$, x_i tendrá los posibles valores $\{0, \dots, p\}$ definidos como:

$$x_i = 0, si\ x_i < \alpha_i^{(1)}$$

$$x_i = 1, si\ \alpha_i^{(1)} < x_i < \alpha_i^{(2)}$$

...

$$x_i = p, si\ x_i > \alpha_i^{(p)}$$

Es decir, la variable x_i tiene valor 0 cuando esta se encuentra por debajo de su primer umbral. En realidad el valor del umbral no es importante; basta con identificar el número de umbrales que podrá tener.

La interacción en el sistema se puede describir mediante ecuaciones lógicas de la forma:

$$X_i(t) = b_i\left(\hat{\boldsymbol{x}}(t)\right), 1 \leq i \leq n$$

en donde el vector $\hat{\boldsymbol{x}}$ denota el *estado lógico* del sistema de regulación, y X_i es la imagen de x_i. La imagen es el valor al cual x_i tiende cuando el estado lógico del sistema es $\hat{\boldsymbol{x}}$. La imagen de x_i no es necesariamente su valor sucesor. La función lógica b_i es una generalización de la función booleana utilizada en el formalismo de redes booleanas, dado que las variables ahora pueden tener más de dos posibles valores.

La función lógica obtiene la imagen de x_i a partir del estado lógico del sistema, más específicamente a partir del valor de k de los n elementos que forman al sistema modelado.

La siguiente figura muestra un ejemplo de red genética modelada bajo el formalismo de redes lógicas generalizadas:

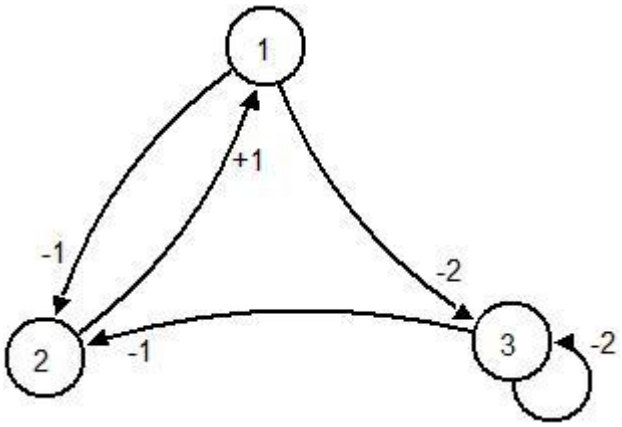


Figura 3.7: Ejemplo de red genética

En la figura anterior, el gen 1 regula al gen 2 y 3, de tal manera que tiene 2 umbrales y por lo tanto la variable correspondiente x_1 toma valores en $\{0, 1, 2\}$. De forma similar, x_2 tiene un umbral y los posibles valores de la variable estarán en $\{0,1\}$ y x_3 tiene dos umbrales por lo que toma valores de en $\{0, 1, 2\}$. Cada arista de la figura se encuentra etiquetada con dos datos. El signo representa, al igual que con las gráficas dirigidas, si se trata de una regulación inhibitoria o no. El número en la arista indica el índice de umbral del que se trata. Por ejemplo, la etiqueta -2 del gen 1 al 3 significa que el gen 1 inhibe al gen 3 por encima de su segundo umbral, es decir, cuando $x_1 = 2$.

La Figura 3.7 puede expresarse mediante las siguientes ecuaciones lógicas:

$$X_1 = b_1(x_2)$$

$$X_2 = b_2(x_1, x_3)$$

$$X_3 = b_3(x_1, x_3)$$

Las funciones b_1, b_2, b_3 se deben especificar de forma consistente con la restricción impuesta por los umbrales en la gráfica de la Figura 3.7.

Las siguientes tablas muestran un ejemplo de posibles funciones lógicas para definir de forma consistente el ejemplo de red genética anterior:

x_2	$b_1(x_2)$
0	**0**
1	**2**

Tabla 3.1: Función lógica para gen 1

x_1	x_3	$b_2(x_1, x_3)$
0	0	**1**
0	1	**1**
0	2	**1**
1	0	**1**
1	1	**0**
1	2	**0**
2	0	**1**
2	1	**0**
2	2	**0**

Tabla 3.2: Función lógica para gen 2

x_1	x_3	$b_3(x_1, x_3)$
0	0	**2**
0	1	**2**
0	2	**1**
1	0	**2**

1	1	**2**
1	2	**1**
2	0	**0**
2	1	**0**
2	2	**0**

Tabla 3.3: Función lógica para gen 3

De las tablas 3.1 a la 3.3, considérese el caso para el gen 2. Si $x_1 > 0$ y $x_3 > 0$, de tal forma que x_1 y x_3 tengan sus valores sobre su primer umbral, entonces las influencias inhibitorias de los genes 1 y 3 se activarán. En la Tabla 3.2 podemos observar que basta con que $x_1 = 0$ o que $x_3 = 0$, es decir, basta con que únicamente una de las dos influencias inhibitorias se inactive para que el gen 2 pueda expresarse aunque sea de forma moderada. En general, se pueden considerar muchas funciones lógicas que sean consistentes con las restricciones impuestas por los umbrales. Exactamente cuál función lógica es elegida dependerá en gran medida de consideraciones de tipo biológico o pueden ser suposiciones derivadas de cierta incertidumbre sobre la topología del sistema que se está modelando.

En este formalismo, un *estado estable* ocurre cuando el estado del sistema es igual al de su imagen:

$$\hat{\boldsymbol{X}} = \hat{\boldsymbol{x}}$$

Para redes pequeñas, encontrar los estados estables puede realizarse de forma exhaustiva mediante pruebas a mano, sin embargo para ejemplos reales esto no es posible sin el apoyo de herramientas computacionales.

De las funciones especificadas mediante las Tablas 3.1 a 3.3, podemos extraer que el único *estado estable* es el estado $[x_1, x_2, x_3] = [2, 1, 0]$ de entre los 18 posibles estados que tiene el sistema descrito. Los 17 estados restantes se llaman *estados transitorios*.

El estado sucesor de un sistema, en términos de este formalismo, debe ser deducido a partir de los posibles valores lógicos definidos por la imagen de cada variable definida en el estado actual del sistema. Para esta deducción, se hace la suposición de que dos variables no cambiarán su valor de forma simultánea, con lo cual se puede reducir de forma considerable el espacio de estados

sucesores. Si un estado es estable, entonces bajo este formalismo se considera que el estado no tiene estados sucesores además de sí mismo, debido a que los valores de sus variables lógicas serán el mismo que el de su imagen.

Este formalismo para modelar redes genéticas se ha implementado y ha demostrado su efectividad en el estudio de ciertos sistemas regulatorios de tamaño limitado. Sin embargo, para sistemas de mayor tamaño la generalización de redes booleanas parece no haber tenido el mismo éxito que las propias redes booleanas sin generalizar.

En la siguiente sección se discute de forma general la elección de un formalismo sobre el otro y las repercusiones que esto conlleva en el estudio de redes genéticas.

3.3 Elección de un formalismo de representación

El verificador que se construyó muestra mediante un enfoque de lógicas temporales la posibilidad de verificar redes genéticas. La elección de un modelo o formalismo para representar redes genéticas no fue parte de nuestro trabajo, sino más bien algo a lo que tuvimos que adecuarnos para poder emplear verificación de modelos. Sin embargo, resulta interesante y de gran valor el justificar el uso de redes booleanas como modelo de representación para los casos de estudio que tuvimos a nuestro alcance. Esta sección se basa principalmente en [11].

Es importante y fundamental la elección de un modelo o formalismo matemático para la representación y verificación de redes genéticas. La elección del tipo de modelo debe hacerse considerando los datos requeridos por el propio modelo para su construcción y los objetivos de modelado y análisis que se quieran alcanzar.

Esta elección involucra pérdidas y ganancias. Por ejemplo, la elección de un modelo más “detallado”, el cual requiera de muchos parámetros, puede ser capaz de capturar información de bajo nivel sobre los fenómenos que ocurren en una red genética. Sin embargo, un modelo de este tipo requerirá grandes cantidades de datos para poder construirlo, resultando en que un modelo con estas características no puede ser construido si la cantidad de datos es insuficiente. Por otro lado, la elección de un modelo “poco detallado” con pocos parámetros será capaz de capturar fenómenos de alto nivel sobre una red genética, como por ejemplo si un gen se encuentra activo o inactivo en un punto determinado del tiempo; para la construcción de este tipo de modelos necesitaremos una cantidad mucho menor de datos, lo cual representa una ventaja considerable.

Este tipo de consideraciones debe llevarnos hacia la elección del modelo o formalismo matemático mediante el cual trabajaremos.

Para la elección de un modelo podemos recordar el principio de la *navaja de Occam:* en ningún caso deberíamos elegir un modelo cuya capacidad sea mayor que lo estrictamente necesario para los objetivos que buscamos satisfacer.

Todo modelo únicamente se aproxima a la realidad por medio de una representación formal. El grado con el cual queremos aproximarnos a la realidad, y más importante, los objetivos que esperamos obtener del modelado para adquirir conocimiento sobre ciertos fenómenos, son parte de lo que determina la elección del modelo.

Desde el punto de vista de redes booleanas como modelos para redes genéticas, discutiblemente su aproximación binaria es excelente. En [11] se estudia si es posible o no extraer información biológica valiosa a partir de datos de expresión genética definidos totalmente en un dominio binario. El estudio busca identificar si cuando los genes se cuantifican a solo dos niveles (1 o 0) serán lo suficientemente informativos para poder separar subclases conocidas de tumores (se trata de un estudio totalmente biológico con repercusiones matemáticas), en cuyo caso se probará que el modelado mediante redes booleanas es realista para representar redes genéticas. Afortunadamente en [11] se menciona que los resultados obtenidos fueron bastante prometedores al usar datos de expresión genética binarios, y empleando la distancia de Hamming como medida de similitud, fueron capaces de mostrar una separación clara entre diferentes tipos de tumores. Esto sugiere que información biológica de valor se puede expresar aún cuando los datos son binarios.

De lo anterior, podemos concluir que para representar la realidad de las redes genéticas es suficiente con utilizar un modelo de representación como lo son las redes booleanas.

La biología utiliza diversas técnicas para la obtención de datos experimentales. Sin embargo, la obtención de datos por medios experimentales es muy susceptible al ruido y otros fenómenos que pueden afectar la reproducibilidad de los resultados. Además, se ha podido observar que desde un punto de vista biológico, la regulación genética exhibe incertidumbre. La evidencia sugiere que esta incertidumbre es ventajosa para algunos mecanismos de regulación. Por lo tanto, desde un punto de vista práctico, las cantidades limitadas de datos con que se cuenta y la naturaleza ruidosa

de las mediciones que se pueden tomar pueden llevar a que la obtención de predicciones sea un proceso complicado bajo ciertos formalismos. De esta manera el uso de un modelado "poco detallado" como los son las redes booleanas se ve justificado. En otras palabras, si nuestro objetivo de modelado fuera capturar las interacciones genéticas a bajo nivel con especial énfasis en los detalles, entonces los datos que se pueden producir con la tecnología disponible en la actualidad no serían adecuados.

Mediante un modelado por redes booleanas podemos obtener información valiosa respecto al comportamiento de las redes genéticas. En general, el interés biológico parece ser el descubrimiento de las relaciones involucradas en la regulación y el control genético, es decir, se busca enfatizar el estudio de propiedades fundamentales en las redes de regulación sin tener que adentrarse en los detalles cuantitativos de cada una de las reacciones involucradas en los diversos procesos biológicos.

Publicaciones recientes indican que un buen número de las preguntas biológicas que se suelen formular para el estudio de redes genéticas pueden ser respondidas a través de un formalismo simplista como lo son las redes booleanas. Tal es el caso del trabajo publicado por el grupo de investigadores con el cual trabajamos y con el cual conformamos los casos de estudio que forman parte de este trabajo.

Todo lo anterior implica que no es necesario emplear modelos mucho más complejos y que la elección de redes booleanas de entre todos los formalismos es adecuado para el tipo de propiedades que se busca verificar.

3.4 Sobre los casos de estudio

Para el desarrollo del verificador de modelos decidimos tomar como caso de estudio el trabajo realizado por el grupo de Elena Alvarez-Buylla del Laboratorio de Genética Molecular del Instituto de Ecología de la UNAM.

La investigación del grupo con el cual interactuamos tiene como objetivo principal la caracterización, construcción y pruebas de un tipo específico de flor, la de la planta *Arabidopsis thaliana*.

En particular, el primer caso de estudio presentado en este capítulo está basado en dos artículos publicados por este grupo de investigación: [12] y [13].

El interés biológico se centra en la especificación de características de regulación e interacción genética; de esto hablaremos como parte introductoria para el primer caso de estudio. Sin embargo, nuestro interés computacional radica en la posibilidad de mostrar el uso de la verificación de modelos como herramienta para obtener resultados sobre el mismo tipo de redes genéticas.

El segundo caso de estudio se presenta aún en términos de un desarrollo nuevo y de una investigación en desarrollo. Se pretende que los resultados que se puedan encontrar respecto al segundo caso de estudio se consigan a través del uso de nuestro verificador de modelos. Para tales fines trataremos la nueva red genética que el grupo de investigación en genética nos propone y estudiaremos los posibles resultados que buscamos encontrar.

3.5 Primer caso de estudio: flor

Como vimos en el capítulo anterior, el tipo de sistema que buscamos modelar es dinámico e involucra genes organizados en forma de una red mediante la cual se generan ciertas regulaciones entre los propios genes. A este tipo de sistemas le llamamos red genética de regulación (GRN por sus siglas en inglés).

En el trabajo publicado sobre la flor de la planta *A. thaliana* y en el cual nos basamos para este caso de estudio se presenta un modelo de red genética de regulación mediante el que se busca entender cómo es que se lleva a cabo la selección de actividad genética en dicha flor.

El enfoque que tomaron en [12] para obtener las reglas de la red, fue el de construir y validar modelos de ciertos procesos biológicos específicos que ya han sido caracterizados. El objetivo es inferir nuevos principios de regulación en células al analizar la dinámica y estructura de modelos específicos de redes genéticas. Se busca que este enfoque complemente los enfoques más teóricos con los que ya se cuenta.

Se espera que el análisis de la estructura y dinámica de redes bien caracterizadas basadas en resultados sólidos experimentales se pueda validar para encontrar huecos o inconsistencias en los

propios datos. Los modelos de redes genéticas de regulación basados en datos experimentales son también importantes para validar inferencias sobre la conectividad de las redes.

En este trabajo, el objeto de estudio son plantas. Las plantas son relativamente más sencillas que los animales en cuanto a su organización celular se refiere. Por lo tanto, parece más realista la posibilidad de proponer modelos de redes genéticas para este tipo de organismos.

La referencia [12] es un trabajo previo y en ella se presentó el modelo de red genética para la flor de *A. thaliana*; en este modelo se propusieron 15 genes. En [12] el modelo que propusieron para representar los datos experimentales con los que se contaba, resultó en un modelo discreto de tres estados. Además, se propusieron reglas lógicas que definían perfectamente la interacción genética; las reglas lógicas propuestas se presentaron en el artículo en forma de tablas. Se considera que la actualización en los valores de activación genética se lleva a cabo de forma síncrona, haciendo posible para los biólogos evitar las posibles interpretaciones subjetivas que pueden derivarse del orden temporal en el cual los genes se pueden o deben actualizar.

En [12], para el estudio del modelo se consideraron 139,968 estados iniciales, a partir de los cuales se alcanzaron únicamente 10 estados finales (tras hacer uso de las reglas lógicas se calculan los estados en el tiempo posterior hasta alcanzar estados estables). Este resultado sugiere que el modelo de regulación genética con el cual se trabajó incorpora componentes clave que permiten validar actividades genéticas que se habían predicho anteriormente por medio de otros modelos.

Sin embargo, desde la publicación de [12] en 2004, el mismo equipo de investigación del Instituto de Ecología encontró ciertos errores menores. La corrección a estos errores y algunas adiciones a dicho artículo se presentan en [13]. En esta nueva publicación se muestra que estos errores encontrados no afectan de ninguna manera el estudio o los resultados antes presentados.

La red genética que se presentó finalmente en [13] puede observarse en la siguiente figura:

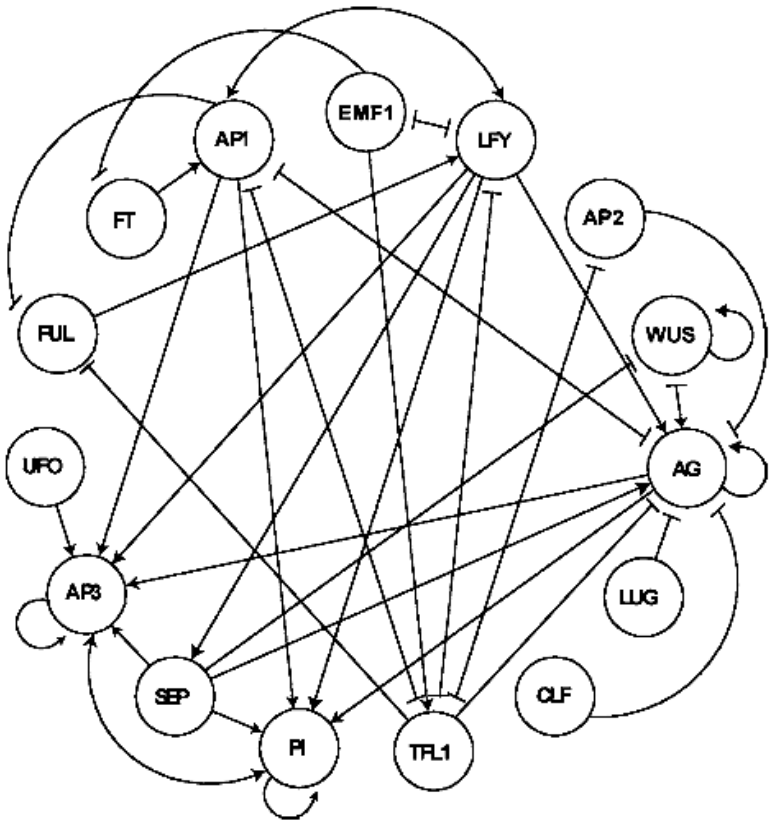


Figura 3.8: Red genética para la flor de $A.thaliana$

En la Figura 3.8, cada nodo (15) corresponde a un gen, y cada arista corresponde a una interacción de regulación, las flechas son regulaciones positivas mientras que las aristas chatas corresponden a regulaciones negativas (inhibición).

En [13] una de las propuestas y actualizaciones principales a la publicación anterior es la reducción de las reglas lógicas de tres a dos estados, con lo cual obtuvieron finalmente una red booleana como modelo para representar la red genética de *A. thaliana.*

La red original contaba con tres estados: inactivo (0), medianamente activo (1) y activo (2). En la red actual se consideró que únicamente debía haber dos estados: inactivo (0) y activo (1). Para la reducción a dos estados, lo que hicieron fue considerar una equivalencia entre el estado intermedio original (medianamente activo) y alguno de los dos estados de la nueva red booleana.

El procedimiento de equivalencia que utilizaron fue el siguiente: Si un estado en [12] había sido considerado como activo (2), en [13] se dejó como activo (1), los estados inactivos se dejaron igual (en 0). En [12] algunas de las reglas lógicas incluían combinaciones de 0-1 y 1-2, en cuyos casos se

tenían las mismas salidas; para estas reglas en [13] se decidió dejar 0 para el primer caso y 1 para el segundo ya que si la salida para 0-1 era la misma entonces no había forma de distinguir entre inactividad y actividad intermedia. La lógica para el segundo caso es similar: si con la combinación de estados de activación 1-2 se obtiene la misma salida en una cierta regla, entonces no hay forma de diferenciar entre activo y medianamente activo, por lo tanto se optó por dejar la regla con valor activo (1). Para las reglas cuya entrada era simplemente 1, optaron por dejar un comodín, en cuyo caso se estaría considerando tanto 0 como 1. Para las salidas de las reglas, los valores se colocaron en base a las nuevas reglas definidas en términos binarios.

Una vez definidas las reglas para la versión binaria de la red genética, la investigación arrojó nuevamente 10 estados estables, o atractores.

Las reglas lógicas que representan la red genética pueden observarse en las Figuras 7.1-7.13 del apéndice. Estas reglas y el formato en el cual las presentamos es el mismo que usamos para nuestro verificador de modelos.

Este caso de estudio tiene un interés particular en el sentido de que permite validar dos cosas al mismo tiempo: el verificador y los resultados obtenidos en [12] y [13]. Por un lado al verificar ciertas propiedades sobre las reglas y compararlas con los resultados publicados en [12] y [13] podemos concluir primeramente si el verificador está funcionando correctamente y al mismo tiempo validar la posibilidad de utilizar una lógica temporal para representar propiedades de sistemas de redes genéticas. Por otro lado, a los grupos de investigación genética les brinda una segunda revisión de sus resultados y finalmente les deja una herramienta que por las propiedades definidas en el capítulo 2 de este mismo trabajo presenta ciertas ventajas sobre las herramientas que utilizan actualmente. Por ejemplo, la posibilidad de trabajar con espacios de estados superiores en tamaño que los que actualmente están trabajando así como la posibilidad, heredada del uso de *CTL* como lenguaje de especificación, de trabajar con redes que presenten incertidumbre, es decir, poder predecir comportamientos sobre redes genéticas aún cuando la información que se posee sobre ciertas interacciones o genes no sea completa.

3.6 Segundo caso de estudio: raíz

Para nosotros el interés del primer caso de estudio radica en la posibilidad de validar el verificador contra resultados ya publicados y aceptados en la comunidad biológica. Al mismo tiempo, estas pruebas nos brindan argumentos para que los biólogos con los cuales estuvimos involucrados se interesen por utilizar una nueva herramienta.

Habiendo mostrado los resultados esperados, los cuales vamos a presentar posteriormente, el grupo de investigación del Instituto de Ecología nos propuso analizar las propiedades genéticas de un nuevo sistema de regulación; nuevo en el sentido de que este grupo no ha realizado trabajos previos respecto a él. Bajo este marco de referencia el segundo caso de estudio tiene como objetivos principales cumplir metas de tipo más bien biológico, se trata de obtener resultados novedosos en el estudio de la raíz de la planta *A. thaliana*.

Con el fin de poder analizar la raíz de la planta en términos de verificación de modelos, la construcción del modelo genético con un enfoque de red booleana está a cargo de los propios biólogos. En este momento nosotros (el equipo de verificación), no contamos con un diagrama de nodos como el que presentamos para la flor de la planta *A. thaliana*, únicamente contamos con algunas de las reglas lógicas que definen la activación genética para la raíz. Es importante destacar que hemos verificado varias versiones de las reglas lógicas para la raíz debido a que conforme se obtienen resultados de la verificación de propiedades, y conforme el grupo de biólogos confirma o desecha ideas, la red va evolucionando. Recordemos también que la red booleana que construyen la realizan en términos de información contenida en literatura previa y en base a experimentos.

En este capítulo únicamente presentamos una de las reglas lógicas con las cuales trabajamos; la regla que presentamos es la que define al gen SCR (Figura 3.9). En el capítulo 5 presentamos los resultados obtenidos tras la verificación de modelos utilizando nuestro verificador. Todas las tablas que conforman la especificación de la raíz de la planta *A. thaliana* las colocamos en el apéndice de este trabajo (Figuras 7.14-7.20); el formato con el cual las presentamos es el mismo que utiliza nuestro verificador.

SCR____SCR SHR JKD MGP WOX5 PLT PIN	
00***00|0	01*0*10|1
00***01|0	01*0*11|1
00***10|0	01*1*00|0
00***11|0	01*1*01|0
01*0*00|1	01*1*10|0
01*0*01|1	01*1*11|0

	10***01|0	1101*01|0
	10***00|0	1101*00|0
10***11|0		1101*11|0
10***10|0		1101*10|0
11*0*01|1		1111*01|1
11*0*00|1		1111*00|1
11*0*11|1		1111*11|1
11*0*10|1		1111*10|1

Figura 3.9: Definición de gen SCR

En la Figura 3.9 podemos observar que el gen SCR es regulado por seis genes (SCR SHR JKD MGP PLT PIN); el gen WOX5 aunque está presente en la tabla se dejó indeterminado. Por lo tanto, se asume que su valor no regula el valor de SCR. La especificación del formato utilizado para las tablas se discute en el capítulo 4.

CAPÍTULO 4
VERIFICADOR

4. 1 Introducción

Este capítulo fue escrito con la intención de generar la documentación necesaria para utilizar el verificador que desarrollamos a manera de Interfaz de Programación de Aplicaciones (*API* por sus siglas en inglés). Se pretende que con ayuda de esta documentación el verificador pueda ser extendido en el futuro por otras personas.

Nuestro verificador fue desarrollado bajo la plataforma *.NET* de Microsoft; específicamente utilizamos el lenguaje de programación C# versión 2.0. Durante la presentación del código en este capítulo suponemos que el lector tiene conocimientos sólidos de programación orientada a objetos y de C#. Como referencia del lenguaje se pueden consultar [15], [16], [17] y [18].

En este capítulo presentamos la arquitectura del verificador de modelos así como una explicación en detalle de los parámetros de entrada para el verificador y de la capa en la cual se implementó el algoritmo de etiquetamiento. Explicamos también, aunque con menos detalle, la implementación de la interfaz gráfica y de las capas de análisis sintáctico y semántico. Durante el recorrido por la implementación del sistema presentamos diagramas de clase que en la mayoría de los casos sirven para dar una vista rápida y general sobre las clases que componen nuestro verificador.

Para lograr mayor portabilidad de nuestro código los nombres de clases, métodos y campos no llevan acentos ni eñes. En el caso de los nombres de campos, utilizamos un guión bajo para facilitar su diferenciación respecto a variables locales que tengan el mismo nombre.

4. 2 Arquitectura del sistema

En esta sección se presenta la arquitectura del sistema; para mejorar el entendimiento se hace uso del Lenguaje Unificado de Modelado (*UML* por sus siglas en inglés) [14].

Al desarrollar el verificador decidimos dividirlo en varios módulos. Esta idea de división modular surgió de la suposición de que el verificador recibirá mantenimiento y en su caso será expandido en trabajos futuros.

El verificador de modelos es una aplicación compleja, que se forma de varios componentes y cada componente lleva a cabo tareas diferentes entre sí. El desarrollo de una aplicación como esta supone la creación de un sistema que cumpla con los requerimientos mínimos que permitan su mantenimiento, reusabilidad, escalabilidad, robustez y seguridad.

Para construir la arquitectura del verificador tomamos en consideración las siguientes ideas:

- La separación modular de componentes nos permite que los cambios que se requieran sobre una cierta funcionalidad no afecten al resto de la aplicación.
- La separación modular de componentes reduce drásticamente la dificultad de la depuración de código (“debugging”) y por lo tanto el tiempo que toma la solución de errores.
- La separación modular de componentes nos permite tener una aplicación escalable. Partiendo del supuesto de que los módulos son intercambiables, no solamente podemos hacer mejoras a los módulos existentes, sino que podremos crear módulos totalmente nuevos para reemplazar los existentes o para añadirlos a la arquitectura.
- La separación modular de componentes permite diversificar el trabajo, de tal manera que en un momento dado podrían conformarse equipos de trabajo y cada uno enfocarse a un módulo del verificador.
- Bajo el esquema de separación modular se fomenta la reutilización de componentes en otros sistemas, i.e. en nuevos desarrollos.
- La arquitectura propuesta debe permitir la planificación y ejecución de pruebas unitarias para finalmente poder comprobar la cohesión de componentes mediante pruebas de integración.

La arquitectura que desarrollamos se dividió en capas. Cada capa lleva a cabo una serie de tareas relacionadas y diferenciadas de las del resto de las capas.

La arquitectura de capas que utilizamos se conoce como un enfoque “relajado” de capas, ya que se permite la comunicación de capas superiores con capas inferiores y no estrictamente con la capa inmediata inferior como sería en un enfoque estricto de capas.

El siguiente diagrama muestra las diferentes capas de las que se compone nuestro verificador de modelos:

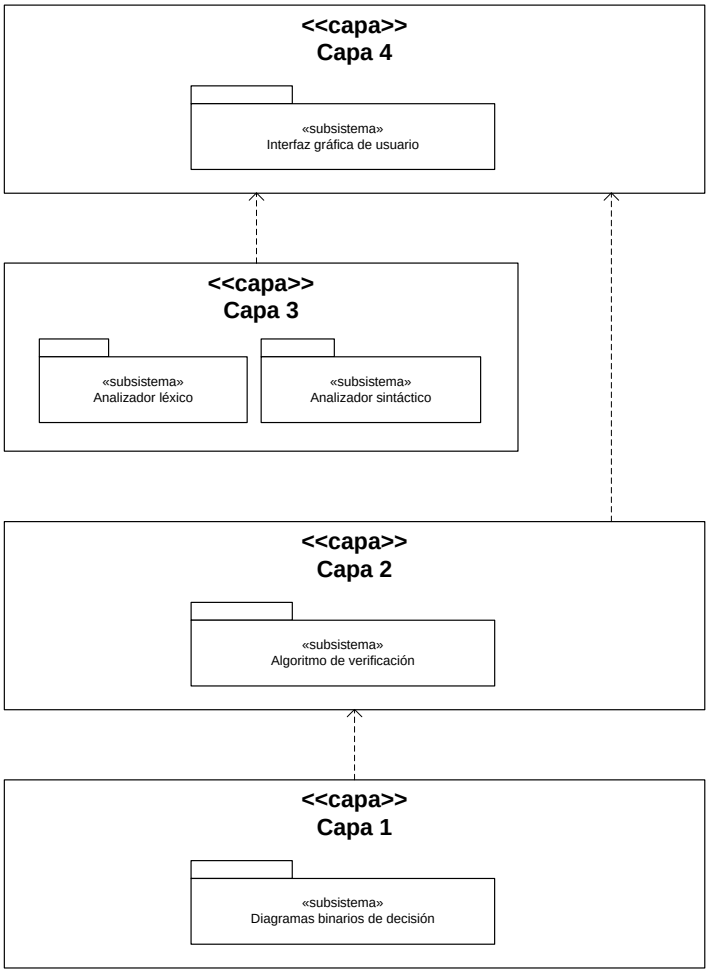


Figura 4.1: Arquitectura del verificador en cuatro capas

Daremos la definición de las diferentes capas más adelante en este mismo capítulo; por ahora basta con que entendamos que el verificador en su conjunto se compone de cuatro capas, cada una con una funcionalidad específica.

Podemos utilizar otra herramienta de *UML* con el afán de que tengamos un entendimiento completo del sistema, en este caso un diagrama de casos de uso. El verificador de modelos es una aplicación muy específica, por lo que los casos de uso parecen ser relativamente pocos. Sin embargo, conviene especificarlos con la suposición de que una definición de este tipo nos facilita modelar el sistema que queremos desarrollar.

En términos generales, el diagrama de casos de uso queda definido como se muestra en la siguiente figura:

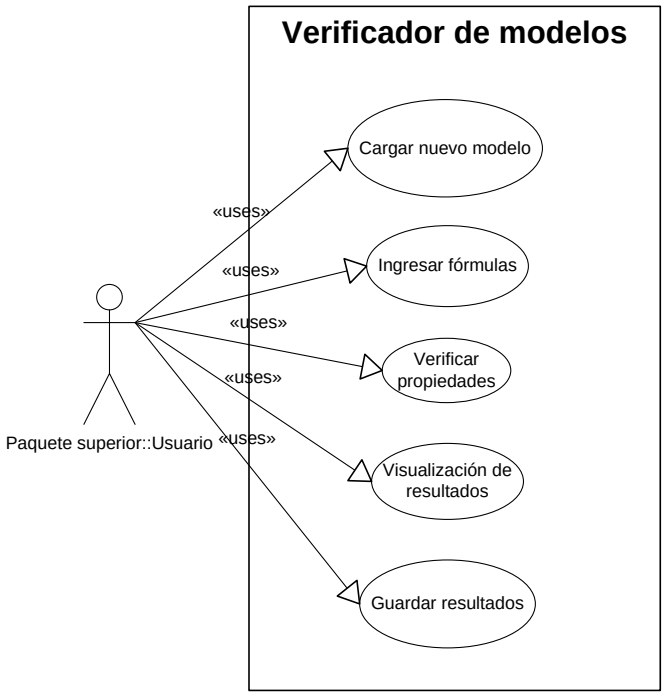


Figura 4.2: Casos de uso de actor Usuario

Podríamos definir más casos de uso y también casos de uso para los propios subsistemas dentro del verificador, como por ejemplo la interfaz (capa 4) hace uso del análisis léxico. Sin embargo, no es nuestro objetivo adentrarnos más en estos detalles.

De la Figura 4.2 podemos ver que en términos generales la interacción que tendrá el usuario con el sistema será la de ingresar modelos para a través de propiedades expresadas en fórmulas obtener un resultado.

Con esto queda definida la arquitectura del verificador. En las secciones que siguen nos adentramos en los detalles de implementación de cada una de las capas; en dichas secciones hacemos uso de otros diagramas, como por ejemplo los diagramas de clases.

4. 3 Especificación de parámetros de entrada

Se podría considerar que el verificador de modelos tiene varios parámetros de entrada. Sin embargo, solo dos son los que nos interesan:

- El modelo.
- Las propiedades.

Consideraremos otros parámetros de entrada, como por ejemplo la elección del formato de presentación de resultados, como parámetros secundarios sin mucha relevancia. Detallaremos dichos parámetros como parte de la explicación en la sección referente a la interfaz gráfica.

Recordemos que el modelo es una representación bajo algún formalismo de un sistema sobre el cual queremos verificar la satisfacción de ciertas propiedades, y recordemos a su vez que estas propiedades que buscamos verificar se escriben en términos de un lenguaje lógico. En nuestro caso ya antes mencionamos que el formalismo para representar el sistema serían redes booleanas y a su vez también expusimos que la lógica a utilizar sería *CTL* con algunas extensiones que nos permiten hacer referencia a estados.

4.3.1 Definición del modelo

Con el objetivo de mantener consistencia con el trabajo tomado para los casos de estudio, decidimos tomar como archivo de entrada para la definición del modelo un formato de tipo tablas de verdad. Este formato tiene además la ventaja de ser uno que ya utilizan en la actualidad el grupo de biólogos con el cual estuvimos trabajando, por lo que no les generamos actividades adicionales en cuanto a la generación de las reglas de la red genética se refiere, y además por obvias razones ellos ya lo entienden.

Por motivos de compatibilidad, el formato elegido para guardar el modelo fue texto plano (*.txt). Dentro de cada archivo se debe ingresar la definición de un solo sistema; esto comprende la definición de todas las tablas que conforman las reglas lógicas de la red.

La sintaxis que utilizamos para las tablas fue heredada de la propia sintaxis que utilizaron los investigadores del Instituto de Ecología en los modelos que nos entregaron para verificación; en la Figura 4.3 puede observarse un fragmento del archivo utilizado para la verificación de la red genética de la flor de *Arabidopsis thaliana*.

```
FUL_____________________AP1 TFL1
    00|1
    01|0
    10|0
    11|0

FT_____________________EMF1
    0|1
    1|0
```

Figura 4.3: Ejemplo tablas en archivo para definición de modelo de Arabidopsis thaliana

La primera tabla en la Figura 4.3 muestra la definición del comportamiento para el gen FUL, en cuyo caso es regulado por los genes AP1 y TFL1. Resulta importante resaltar que los encabezados de las tablas muestran primero el gen siendo definido y posteriormente los genes que lo regulan, mientras que la propia tabla de verdad (los unos y los ceros) muestra primero los genes que regulan y posteriormente el gen siendo definido. Aunque esto en un principio resultó poco natural, decidimos mantenerlo así con el objetivo de mantener el uso de los archivos generados por los propios biólogos.

En términos de Backus Naur Form (*BNF*), podemos definir la sintaxis para el encabezado de la siguiente manera:

$$Encabezado ::= Salida\ Separador\ Entradas$$

$$Salida ::= p$$

$$Separador ::= \_\_\ Separador \mid \_\_$$

$$Entradas ::= p\ Entradas \mid p$$

En donde p es un átomo de un conjunto *Átomos* de las variables proposicionales que ocurren en el modelo. En particular, para los casos de estudio considerados, p será alguno de los genes que definen la red genética.

En el caso del cuerpo de la tabla, se puede considerar la siguiente definición en *BNF* para cada renglón:

$$Renglón ::= Entradas\ Separador\ Salida$$

$$Entradas ::= 0\ Entradas \mid 1\ Entradas \mid * Entradas \mid 0 \mid 1 \mid *$$

$$Separador ::= "|"$$

$$Salida ::= 1 \mid 0 \mid *$$

En el caso de $Separador$, la barra es un símbolo del alfabeto y no del metalenguaje para definir el renglón. Por esa razón lo colocamos entre comillas.

Con la definición anterior para cada renglón, podemos resaltar la existencia del comodín $*$ como parte de los posibles símbolos de la $Salida$ y de las $Entradas$. Esto nos permite especificar modelos indeterminados, en cuyo caso la presencia de un $*$ como salida de un renglón, se interpretará como la existencia de cuando menos dos estados sucesores en lugar de un único estado sucesor. En capítulos anteriores ya discutimos sobre la ventaja de poder especificar modelos no determinados y sobre el hecho de que bajo un algoritmo basado en *CTL* esto no representa un problema.

Para aclarar el uso del comodín y la definición del archivo de entrada para el verificador, podemos utilizar el siguiente sistema:

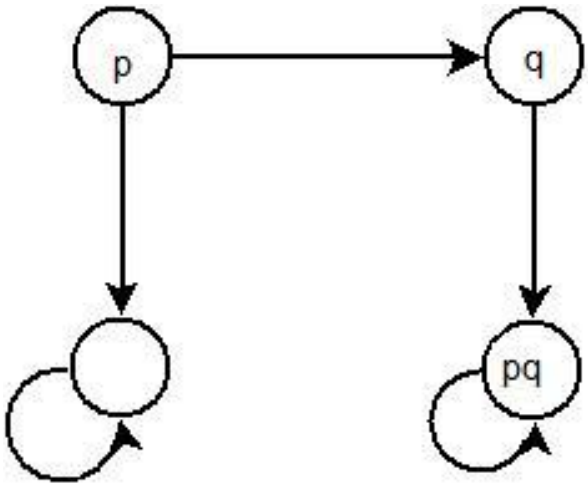


Figura 4.4: Ejemplo de sistema

En este caso la Figura 4.4 puede ser representada únicamente si hacemos uso de la sintaxis con comodín. Las tablas de verdad resultantes quedarían como se muestra a continuación:

p_____p q
00|0
01|1
10|0
11|1
q__p q
00|0
01|1
10|*
11|1

Figura 4.5: Tabla para la figura 4.4

En la figura anterior puede observarse el uso del comodín para la definición de q, en cuyo caso tenemos que bajo las entradas $p \wedge \neg q$ y dado el sistema de la Figura 4.4 (corresponde al estado superior izquierdo) no existe un único estado siguiente (existen dos); bajo estas condiciones sin comodín sería imposible representar mediante tablas el sistema de ejemplo.

Para más ejemplos simples de sistemas con sus respectivos archivos de entrada puede consultarse el apéndice (Figuras 7.21-7.28).

4.3.2 Definición de propiedades

Igual de importante que la definición del modelo es la definición de propiedades, ya que estas especifican las características del sistema que buscamos verificar. Es por ello necesario entender cabalmente el lenguaje utilizado para especificarlas. Para nuestro verificador la definición de propiedades se hace mediante el uso de *CTL* híbrido; para información respecto a *CTL* híbrido consultar la sección 6 dentro del capítulo 2 de este mismo trabajo.

4.4 Verificador de consola

Una de las capas más importantes dentro del verificador de modelos es por supuesto el propio algoritmo de verificación. En nuestro caso encapsulamos el algoritmo de verificación dentro de la segunda capa de la arquitectura; esta capa se compone básicamente de un solo subsistema.

Al diseñar el componente de verificación decidimos que sería esencial, además de permitir su interacción con otros componentes para formar parte de la arquitectura que proponemos, también permitir su ejecución de forma independiente de forma que pudiéramos probarlo y al mismo tiempo generar un núcleo de verificación ligero y portable. Con este objetivo el algoritmo de verificación se encuentra encapsulado en un proyecto de tipo consola, el cual puede ser ejecutado y sin presentar una interfaz rica en contenido se pueden llevar a cabo básicamente las mismas tareas que se permiten mediante la interfaz gráfica. Obviamente el uso del verificador desde la consola tiene desventajas contra el uso de la interfaz gráfica. Por ejemplo, no tenemos accesorios intermedios como los analizadores léxicos y sintácticos, con lo cual debemos utilizar un lenguaje intermedio para la definición de propiedades. Existen otras diferencias, estas se harán visibles cuando tratemos el tema de la interfaz gráfica.

Para poder explicar este subsistema, comenzaremos presentando un diagrama de clases sin detalles, esto nos permite tener una vista "aérea" del verificador de consola:

Figura 4.6: Diagrama de clases sin detalles

Como se puede observar en la figura, el verificador de consola se compone de cinco clases normales, una clase estática, una interfaz y una enumeración; haremos evidentes en secciones subsecuentes la razón y el uso de cada una de estas entidades.

Conforme vayamos avanzando en la explicación de las entidades que componen este subsistema iremos expandiendo el diagrama de clases de tal forma que podamos tener en cada ocasión una visión general y posteriormente una explicación sobre la implementación.

4.4.1 Procesamiento del archivo de especificación

La representación del sistema a verificar sobre la cual trabaja el verificador de modelos (diagrama binario de decisión) se construye a partir de un modelo especificado mediante un archivo de texto plano siguiendo las reglas establecidas en la sección 3.1 de este capítulo para su elaboración. Para la construcción de la representación sobre la cual trabaja el verificador se requiere por supuesto de procesos que permitan llevar a cabo la lectura del archivo de entrada, así como la construcción del respectivo diagrama binario de decisión a partir de las tablas que especifican la interacción de nodos. Para tales fines generamos varias entidades dentro del verificador de tal forma que podamos probarlas unitariamente y al mismo tiempo promovemos la separación en entidades independientes que, como ya discutimos antes, beneficia el mantenimiento del componente.

En la cima de las entidades que nos permiten realizar estas tareas se encuentra la interfaz *ArchivoEstructura*. El diagrama para dicha interfaz es el siguiente:

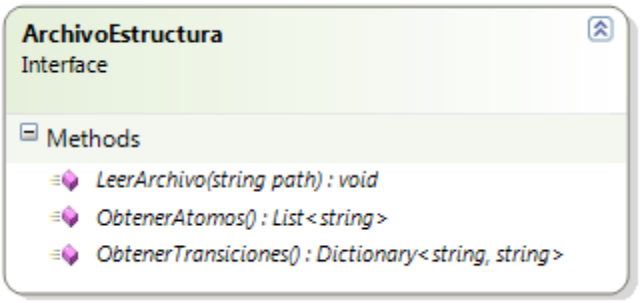


Figura 4.7: Interfaz ArchivoEstructura

El algoritmo que creamos para la lectura de la tabla de los biólogos basa su funcionamiento en la interfaz mostrada en la figura anterior. Gracias al uso de este tipo de entidades podemos modificar fácilmente la lectura de tal forma que se permitan diferentes estructuras de representación (no únicamente las tablas como las tenemos definidas ahora). Al mismo tiempo se establece el mínimo conjunto de funcionalidades que deberá programarse para la lectura de archivos y su futura interacción con el algoritmo de verificación.

La interfaz *ArchivoEstructura* es una representación de alto nivel de los archivos que pueden cargarse en el verificador. Al igual que el resto de las entidades del verificador de consola, esta interfaz se encuentra anidada dentro del espacio de nombres que hemos llamado *Consola_Verificador*.

Como parte de los métodos que se deberán implementar se encuentran la lectura de archivos que recibe como parámetro de entradas una ruta (o "path" en inglés) en la cual encontrar el archivo con la especificación del modelo. Asimismo, se deberá especificar un método que permita obtener una lista genérica de cadenas mediante la cual sea posible recuperar la lista de átomos definida en el archivo así como un diccionario (en terminología de C# es una tabla de "hash" genérica), compuesto de una cadena como llave y cuyo valor almacenado también será una cadena, mediante el cual podamos recuperar las transiciones definidas en el archivo de entrada.

En general a lo largo de este capítulo solo mostraremos fragmentos de código cuando consideremos que su presentación puede servir para aclarar detalles de implementación. En este caso el código para la interfaz *ArchivoEstructura* resulta interesante para generar una analogía con la Figura 4.7 y familiarizar aún más al lector con los diagramas que presentamos:

```
interface ArchivoEstructura
{
    void LeerArchivo(string path);
    Dictionary<string, string> ObtenerTransiciones();
    List<string> ObtenerAtomos();
}
```

Figura 4.8: Código para interfaz ArchivoEstructura

Si comparamos las Figuras 4.7 y 4.8 podemos observar la relación existente entre código y diagrama. En particular en el caso de la interfaz es normal no encontrar la implementación de los métodos; en el caso de las clases los métodos tendrán implementación y los diagramas se verán similares al diagrama para la interfaz.

Siguiendo un enfoque jerárquico, la siguiente entidad que se debe discutir es la propia implementación de la interfaz *ArchivoEstructura*, en este caso la clase que creamos para dicho objetivo la llamamos *TablaBiologos*, el diagrama para esta clase se puede observar en la Figura 4.9.

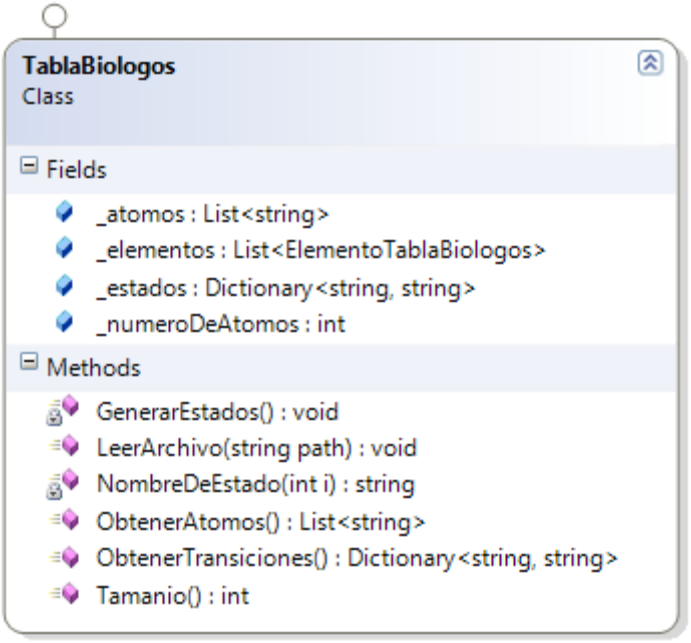


Figura 4.9: Clase TablaBiologos

La clase *TablaBiologos* almacena todos los átomos (en particular para los casos de estudio, los genes) en forma de una lista genérica de cadenas. Durante la lectura de la especificación (desde el archivo plano) se valida si la lista genérica de átomos contiene o no el átomo siendo procesado. En caso negativo (si la lista no lo contiene), se agrega el nombre del átomo a la lista.

En el caso de los estados que conforman el modelo, lo que utilizamos fue un diccionario por dos razones principales. La primera es que siendo un diccionario podemos colocarle a los estados una llave y un valor desde la propia implementación de la estructura. La segunda es que el hecho de manejar diccionarios para los estados nos permite su fácil recuperación a través de la llave, es decir, nos evitamos hacer búsquedas para conocer los valores almacenados en un estado dado. Nosotros decidimos almacenar como llave el nombre del estado actual, y como valor el nombre del estado siguiente al estado actual (el cual puede o no tener comodines). Como nombre de estado consideramos los valores de verdad (1, 0) para los átomos en dicho estado. El procedimiento que utilizamos para generar los nombres puede observarse en la Figura 4.10.

```
private string NombreDeEstado(int i)
{
    string binario = Convert.ToString(i, 2);
    long entero = Convert.ToInt64(binario);
    return entero.ToString("D" + _numeroDeAtomos);
}
```

Figura 4.10: Procedimiento para nombrar estados durante la lectura del archivo de entrada

El procedimiento anterior utiliza las clases de conversión de *.NET*. En general lo que hacemos es convertir el entero que se recibe a su representación en binario y finalmente completar los ceros restantes hasta cubrir una longitud igual al número de átomos que tenga el modelo que se está construyendo. El objetivo es homogeneizar a la misma longitud el nombre de todos los estados al mismo tiempo que los dejamos a todos en términos binarios.

Durante la construcción de los estados que conforman el modelo, inicialmente consideramos que todos los estados siguientes se encuentran no definidos, y conforme vamos leyendo el archivo de entrada y vamos asignando nombres a los distintos estados vamos también definiendo las salidas como 1, 0 o * en caso de salidas que por especificación no están definidas.

Algo importante dentro de los archivos de definición es la existencia de diferentes tablas de verdad. En cada tabla se definen un conjunto de variables de entrada y una variable de salida. Durante el procesamiento del archivo resultó conveniente definir una estructura que nos permitiera representar cada una de estas tablas. La estructura que generamos se implementó en forma de clase que a su vez se diseñó para ser consumida por la clase con el procesamiento del archivo. En la Figura 4.9 puede observarse que la lista genérica *_elementos* almacena justamente un tipo de dato creado por nosotros, en este caso objetos de tipo *ElementoTablaBiologos*. Estos objetos son una abstracción de las tablas de verdad. Daremos más adelante en este misma sección la discusión sobre la implementación de esta clase.

En términos generales el método directriz dentro de la clase *TablaBiologos* es *LeerArchivo*. Los métodos *ObtenerAtomos* y *ObtenerTransiciones* funcionan como métodos accesores, en ambos casos como métodos *GET*. El método accesor *ObtenerAtomos* regresa el contenido del campo de clase *_estados*, mientras que *ObtenerTransiciones* regresa *_atomos*.

El diagrama para la clase *ElementoTablaBiologos* que utilizamos para almacenar las diferentes tablas de verdad es el siguiente:

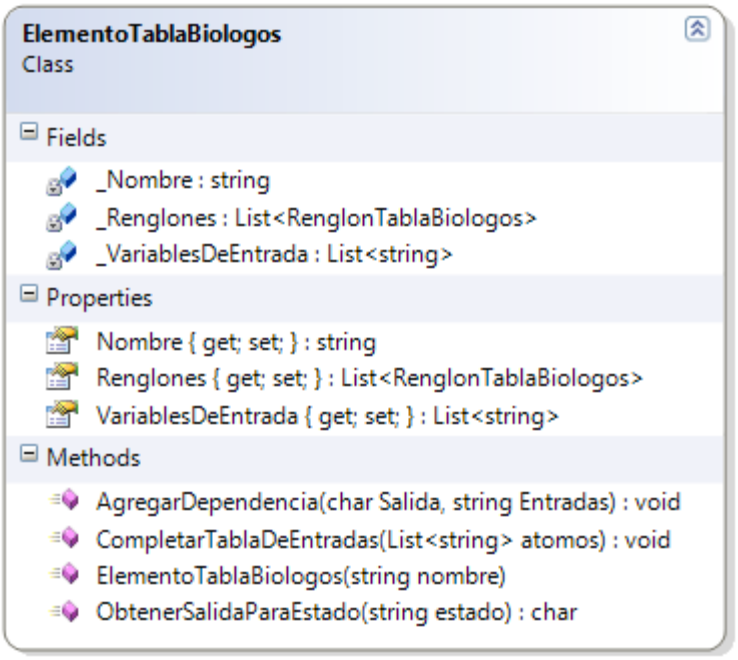


Figura 4.11: Clase ElementoTablaBiologos

Los objetos de tipo *ElementoTablaBiologos* se construyen a través de un nombre en forma de cadena. En este caso el nombre que le asignamos a cada tabla corresponde a la variable de salida para dicha tabla (el gen siendo definido). Esta asignación se realiza dentro del método *LeerArchivo* de la clase *TablaBiologos* que es en donde se crean las instancias de *ElementoTablaBiologos*.

Cada tabla de verdad en la especificación del sistema contiene una serie de renglones binarios en los cuales se define explícitamente la interacción existente entre los diferentes átomos (para nosotros la regulación genética). Una vez más resulta conveniente abstraer la definición de un renglón y generar una estructura de datos para almacenarlo; como parte de la implementación de las tablas de verdad utilizamos una lista genérica que almacena renglones bajo la estructura especificada en los archivos de entrada. Más adelante daremos la explicación sobre esta implementación.

En las tablas de verdad especificadas en los archivos de entrada puede observarse que aun cuando el sistema se compone de n átomos, frecuentemente las tablas se definen en términos de k

átomos en donde $k < n$. Para poder trabajar con estos sistemas es necesario completar dichas tablas. Cuando una tabla de verdad no define en sus entradas $n - k$ átomos, y estos átomos sí forman parte del sistema, la interpretación que damos es la de un término (conjunción de átomos posiblemente negados, i.e. literales) en el que no ocurren todos los átomos: Los átomos ausentes pueden tomar cualquier valor de verdad. Esto equivale a poner un comodín en el valor de dichos átomos. Esto lo conseguimos a través del método *CompletarTablaEntradas* (puede observarse la definición de parámetros en la Figura 4.11); este método se invoca en la clase *TablaBiologos* dentro del método *GenerarEstados* antes de la asignación final de valores.

Internamente la clase *ElementoTablaBiologos* utiliza extensivamente la definición de renglón. Por ejemplo, el método *AgregarDependencia* únicamente agrega un nuevo renglón a la tabla; esto puede observarse en la siguiente figura:

```
public void AgregarDependencia(char Salida, string Entradas)
{
    _Renglones.Add(new RenglonTablaBiologos(Salida, Entradas));
}
```

Figura 4.12: Método AgregarDependencia

En la Figura 4.11 observamos por primera vez una sección denominada Propiedades (*Properties*); las propiedades en C# representan métodos accesores. En el caso de la Figura 4.11, las tres propiedades son de tipo *GET* y *SET*, cada una de estas propiedades permite el acceso a uno de los campos definidos dentro de la clase.

Las instancias de la clase *RenglonTablaBiologos* son múltiples dentro de la definición de cada tabla de verdad. El diagrama para la clase *RenglonTablaBiologos* puede observarse en la Figura 4.13.

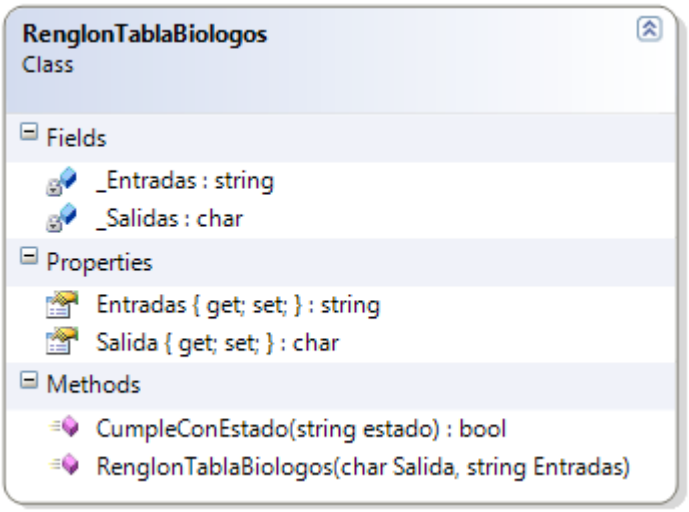


Figura 4.13: Clase RenglonTablaBiologos

La clase *RenglonTablaBiologos* se compone de dos campos. En uno de los campos se almacenan las entradas; el formato que utilizamos para almacenar las entradas en un renglón dentro del verificador es el mismo formato que se utiliza en los archivos de especificación: binario. En el otro campo se almacenan las salidas del renglón. En el caso de las salidas fue necesario un campo que permitiera un carácter único pues la definición de la salida para cada renglón en los archivos de especificación es para un único átomo (gen) de forma tal que solo necesitamos un bit de almacenamiento; en este caso el campo *_Salidas* se llena únicamente con 0 o 1.

Las propiedades en la clase *RenglonTablaBiologos* nos permiten hacer validaciones sobre las entradas y salidas de las tablas de verdad, con lo cual conseguimos mayor seguridad en la implementación de la estructura de datos para representación de modelos. En este caso el accesor *SET* tanto de entradas como salidas puede utilizarse también para hacer una evaluación de contenido. Realizamos la evaluación de contenido fácilmente mediante el uso de expresiones regulares del tipo $(0 + 1 + *)$ para la salida (en donde $*$ es un símbolo del alfabeto) y podemos utilizar expresiones similares a la anterior para las entradas. Podemos utilizar la clase Regex (para expresiones regulares) de *.NET* en conjunto con la definición anterior de expresiones regulares de tal forma que la validación de entradas es sencilla. Como puede apreciarse, organizamos la validación en componentes internos que claramente son más simples que los componentes externos que llevan a cabo tareas como la propia lectura del archivo. Esto resulta en pruebas más simples de código.

En la Figura 4.13 podemos observar la existencia de otro método además del constructor de clase. El método *CumpleConEstado* recibe un estado en forma de cadena (valores binarios que identifican al estado) y regresa un valor booleano. Consumimos el método *CumpleConEstado* dentro de la clase *ElementoTablaBiologos* para obtener la salida de un estado determinado en el modelo. El método *CumpleConEstado* compara el estado para el cual buscamos su salida con un tipo de dato renglón (el cual puede estar compuesto por `0`, `1` o `*`). La comparación busca equivalencia entre las entradas del renglón y el estado. Si las entradas coinciden con el estado, podemos regresar la salida especificada en el archivo de entrada. En cambio, si el estado para el cual queremos obtener su salida no se encuentra definido en la colección de renglones, sabemos implícitamente que su salida debe ser 0 (bajo la suposición de que cualquier activación en el modelo debe ser explícitamente definida en la red booleana).

Al diseñar las clases y métodos que componen el procesamiento del archivo de especificación (archivo con todas las tablas para la definición del modelo) se consideró el formato de redes booleanas. Sin embargo, como parte del análisis y los objetivos alrededor de la creación de un verificador siempre tomamos en cuenta que su uso pudiera ser no exclusivo de la biología. Esa es la razón principal por la cual todo este procesamiento se ve tan segmentado en entidades. En la base de este procesamiento observamos una interfaz. La justificación primaria de implementar (que es diferente a heredar en programación orientada a objetos) una interfaz como primer paso en la construcción del procesamiento es la de generar un esquema de lectura/carga de archivos que pudiera ser tan independiente del formato mismo del archivo de entrada como fuera posible. La facilidad con la cual podemos redireccionar el verificador hacia nuevas áreas de estudio es enorme: las clases para procesar archivos se pueden intercambiar o mejorar con toda libertad. La única regla que debemos considerar de forma constante es que cualquier nueva implementación de clases o mejora a las ya existentes (en cuanto al procesamiento del archivo de entrada se refiere) debe ser una implementación de la interfaz *ArchivoEstructura*. Mediante esta única regla podemos asegurar que el resto del verificador podrá interactuar con cualquier nueva representación que podamos utilizar para especificar los sistemas en los archivos de entrada. Por lo tanto, hay que resaltar que aunque en un principio nuestro verificador es para verificar redes genéticas, en trabajos futuros no será complicado verificar cualquier otro sistema que sea susceptible de ser modelado.

4.4.2 Diagramas binarios de decisión mediante CUDD

En la sección anterior discutimos acerca de cómo diseñamos e implementamos los procesos para cargar el modelo a partir de archivos de texto plano mediante las clases e interfaz que implementamos para cumplir con dicho objetivo. Una vez concluido el procesamiento del archivo de entrada, entiéndase una vez que hemos leído las tablas y conocemos todos los átomos que conforman al modelo y por lo tanto tenemos toda la información en memoria principal, podemos comenzar la generación del respectivo diagrama binario de decisión. En esta sección vamos a presentar cómo llevamos a cabo la generación de los diagramas binarios de decisión y presentamos también de forma técnica cómo invocar las funciones básicas sobre estos diagramas. Para conocer más sobre el funcionamiento interno de estas funciones consultar el capítulo 2 de este trabajo.

Los diagramas binarios de decisión son una estructura de datos que ya se ha estado utilizando desde hace tiempo con el objetivo de poder representar hasta trillones de estados [9]. Por lo tanto, en lugar de plantearnos el objetivo de crear nuestra propia implementación de estos diagramas resulta conveniente voltear la mirada hacia implementaciones de estos diagramas que ya han sido probadas y han demostrado su efectividad. Existen varias implementaciones de diagramas binarios de decisión que son libres de descargar e incluso modificar. Algunos de los paquetes que evaluamos antes de tomar una decisión para nuestro verificador son:

- *BuDDy* [M5]: paquete de diagramas binarios de decisión, implementa reordenamientos dinámicos, recolección automática de basura, está implementado en C/C++.
- *JDD* [M6]: paquete de diagramas binarios de decisión inspirado en *BuDDy*, implementa únicamente operaciones básicas, está implementado en Java.
- *CUDD* [M3]: paquete de diagramas binarios de decisión, implementa las principales funciones sobre *BDDs*, su diseño permite su utilización en aplicaciones de terceros mediante la exportación de funciones, está implementado en C/C++.

Es obvio que lo que buscamos es un paquete que pueda ser adaptado para nuestro verificador. No existe una implementación de diagramas binarios de decisión en C#; prácticamente todo lo existente es para C/C++ o Java.

En un principio evaluamos utilizar *JDD* pues al estar totalmente implementado en Java resulta fácil su migración hacia C#. Sin embargo, *JDD* tiene algunas restricciones que en el mismo sitio del paquete se mencionan. Entre las más importantes se encuentran el ineficiente manejo de memoria. Encontramos un problema aún mayor con este paquete dentro del propio código fuente, en donde por la misma restricción de la memoria se tiene una cota en el número de estados que puede tener el diagrama.

El mismo autor (Arash Vahidi) que implementó *JDD* también desarrolló una interfaz para consumir *CUDD* y *BuDDy* mediante un precompilado con Java. Esta interfaz tiene el nombre *JBDD*. *CUDD* es utilizado en otros verificadores de modelos existentes en el mercado como por ejemplo *NuSMV* y ha demostrado su efectividad para almacenar sistemas con un gran número de estados.

Elegimos trabajar con *CUDD* a través de la interfaz *JBDD*. Para poder interactuar con esta interfaz diseñada para Java hicimos uso de *IKVM*, una implementación de la máquina virtual de Java para Mono y Microsoft .NET Framework. Esta implementación incluye, además de la máquina virtual, la implementación de las clases de Java para *.NET* y una serie de herramientas que nos permiten la interoperabilidad necesaria entre Java y *.NET*. *JBDD* incluye un precompilado de *CUDD* en forma de librería; este código encierra toda la implementación.

Los archivos de *IKVM* necesarios para utilizar *JBDD* en *.NET* se listan en la siguiente tabla:

Archivo	**Propósito**
ikvm-native.dll	Implementación de algunas partes de la interfaz *JNI*
IKVM.Runtime.dll	La maquina virtual de Java
IKVM.OpenJDK.ClassLibrary.dll	Versión compilada de la librería de clases de Java

Tabla 4.1: Archivos necesarios para consumir JBDD

Los archivos listados en la Tabla 4.1 permiten consumir la interfaz (“wrapper”) de *JBDD* mediante *.NET* (C# en nuestro caso). Para generar la interfaz de *JBDD* para *.NET* se debe compilar *JBDD* (de Java) como un archivo *JAR* para finalmente hacer la conversión a “assemblies” de *.NET* (dll). Para el caso en que fuera necesario actualizar la versión de *CUDD* existente hemos colocado en el

apéndice de este trabajo los comandos que se deben ejecutar con el fin de obtener la *DLL* que podemos consumir en *.NET*.

A través de la interfaz *JBDD* tenemos acceso a las funciones principales de *CUDD*. Entre las funciones que podemos invocar se encuentra una implementación del algoritmo $exists$. En tratamientos de diagramas binarios de decisión (p.ej. secciones 2.8.1 y 2.8.2) es común dar una metafunción genérica $apply$ que se instancia con el operador booleano apropiado (i.e. conjunción, disyunción, implicación, etc.). Sin embargo, en *JBDD* cada operador booleano está implementado por separado como una primitiva.

CUDD internamente guarda los nodos del diagrama binario mediante variables de tipo entero. Sin embargo, manipula estas variables a nivel de bits mediante los operadores binarios de C. Externamente a través de *JBDD* únicamente podemos observar valores enteros. Por lo tanto, si queremos identificar los nodos que conforman nuestro diagrama mediante sus nombres, debemos implementar alguna estructura externa al paquete de diagramas. En nuestro verificador, con el fin de poder recuperar cualquier nodo del diagrama, implementamos una clase llamada *nodoBDD*. Para entender el funcionamiento de la clase *nodoBDD* presentamos su diagrama de clase en la Figura 4.14:

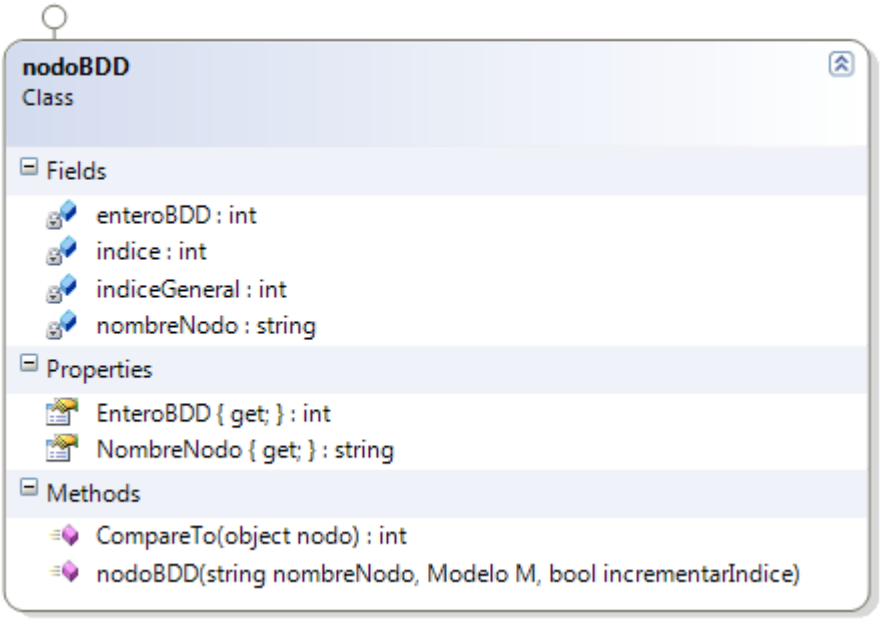


Figura 4.14: Clase nodoBDD

En la Figura 4.14 podemos observar que colocamos un campo de clase llamado *enteroBDD* de tipo entero. El objetivo de este campo es almacenar el valor que *CUDD* le asigna a un nodo. En el mismo objeto de tipo *nodoBDD* almacenaremos un nombre significativo (el nombre de un estado en el sistema, por ejemplo) que nos permita recuperar fácilmente el valor mediante el cual se guardó dentro del diagrama binario de decisión.

Podemos observar en la Figura 4.14 que la clase *nodoBDD* implementa una interfaz, en este caso la interfaz *IComparable*. La interfaz *IComparable* es propia de *.NET*. Al implementar *IComparable* en la clase *nodoBDD* nos vemos obligados a dar la implementación para el método *CompareTo*. Una clase que implementa *IComparable* es una clase que puede ordenarse mediante los algoritmos de ordenamiento propios de las colecciones dentro de *.NET*; podemos meter objetos de tipo *nodoBDD* en colecciones, por ejemplo en listas genéricas, y hacer uso de los algoritmos de ordenamiento propios de dichas colecciones para ordenar de acuerdo al criterio implementado en nuestro método *CompareTo.* La implementación en la clase *nodoBDD* para el método *CompareTo* se encuentra en la Figura 4.15:

```
public int CompareTo(object nodo)
{
    nodoBDD otroNodo = nodo as nodoBDD;
    if (otroNodo != null)
    {
        if (this.indice < otroNodo.indice)
            return -1;
        else if (this.indice > otroNodo.indice)
            return 1;
        else
            return 0;
    }
    else
    {
        throw new InvalidCastException("tipo inválido para
         comparación");
    }
}
```

Figura 4.15: Método CompareTo

Nuestra implementación de *CompareTo* es robusta y segura debido a que recibimos como parámetro de entrada objetos y hacemos la conversión ("cast" en inglés) de tipo de forma segura mediante el operador *as*, posteriormente validamos la conversión y en caso de encontrarnos con objetos erróneos lanzamos una excepción de tipo conversión. Como puede observarse en la Figura 4.15 el parámetro de comparación que usamos para el ordenamiento es un índice; no ordenamos respecto al entero generado por el diagrama ni al nombre del nodo. El campo privado *indice* es un contador que vamos incrementando conforme se van creando nodos; de esta manera cada nodo que se crea en el sistema obtiene un identificador único. El ordenamiento mediante orden de creación nos permite identificar fácilmente los nodos actuales con los nodos siguientes en el sistema durante la ejecución del algoritmo de etiquetamiento, i.e. sus índices coinciden.

El constructor de la clase *nodoBDD* recibe entre sus parámetros un objeto de tipo *Modelo*. Dentro de la clase *Modelo* es donde creamos el objeto que nos permite la comunicación con *JBDD* y finalmente con *CUDD*. Podemos ver en la Figura 4.16 el diagrama de la clase *Modelo*. El objeto de tipo *Modelo* es necesario para crear el espacio de memoria dentro del diagrama binario de decisión.

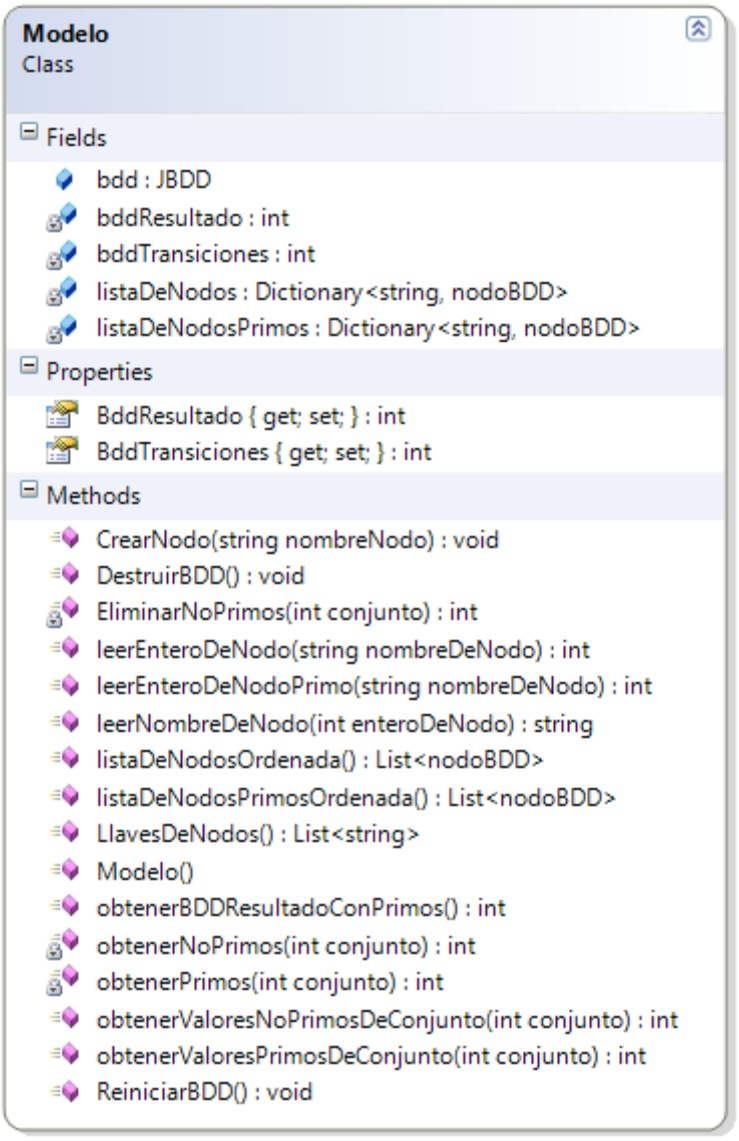


Figura 4.16: Clase Modelo

Entre los campos de la clase *Modelo* tenemos uno de tipo *JBDD*. El campo *JBDD* se inicializa a nivel de clase. En el constructor de la clase *Modelo* inicializamos el campo *bddResultado* como se muestra en la Figura 4.17. El campo *bddResultado*, al cual tenemos acceso desde el exterior mediante el accesor *BddResultado,* es de tipo entero. El valor entero que se asigna dentro del campo *bddResultado* se genera y se modifica mediante operadores propios del diagrama binario conforme el algoritmo de etiquetamiento se ejecuta. A través de *bddResultado* tenemos acceso al diagrama binario de decisión que representa los estados que satisfacen la propiedad que hayamos verificado.

```
public Modelo()
{
    this.BddResultado = this.bdd.getZero();
}
```

Figura 4.17: Constructor de la clase Modelo

En la Figura 4.17 observamos una invocación a la función *getZero*; la función *getZero* es una función propia de *CUDD*, la cual nos permite obtener el diagrama binario de decisión que representa la constante cero. La constante cero representa el conjunto vacío.

La administración de la memoria cuando trabajamos con diagramas binarios de decisión es importante. Es por ello que programamos un par de métodos que contienen las directivas necesarias para reiniciar al paquete de diagramas *CUDD*. Antes de terminar una ejecución que implique diagramas, debemos destruir el *BDD* mediante el método *DestruirBDD* a través de un objeto de tipo *Modelo*. Si vamos a reiniciar la verificación, es conveniente utilizar el método *ReiniciarBDD*.

En la clase *Modelo* utilizamos a nivel de clase dos diccionarios genéricos. Utilizamos el diccionario genérico *listaDeNodos* para almacenar los nodos actuales. Como llave del diccionario empleamos el nombre del nodo y como valor un objeto de tipo *nodoBDD* con todas las propiedades que ya discutimos antes. El diccionario genérico *listaDeNodosPrimos* tiene la misma estructura que el diccionario *listaDeNodos*, pero en esta ocasión en este diccionario almacenamos los nodos siguientes al nodo actual. La importancia del campo *indice* dentro de la clase *nodoBDD* se encuentra en estas listas. En la clase *Modelo* tenemos el método a través del cual generamos nodos en el diagrama binario (Figura 4.18). Cuando construimos un nodo, generamos al mismo tiempo el nodo actual y el nodo siguiente. En el caso del nodo actual indicamos al objeto de tipo *nodoBDD* que incremente el índice de creación; en el caso del nodo siguiente le indicamos al objeto *nodoBDD* que adquiera el mismo índice de creación que se asignó para el nodo actual. Si deseamos emparejar los nodos actuales con su nodo siguiente únicamente necesitamos recuperar ambos diccionarios genéricos de forma ordenada con base en el valor almacenado, es decir, ordenamos con base en los objetos *nodoBDD* que guardamos dentro del diccionario.

```
public void CrearNodo(string nombreNodo)
{
    nodoBDD nuevoNodo = new nodoBDD(nombreNodo, this, true);
    listaDeNodos.Add(nombreNodo, nuevoNodo);

    nodoBDD nuevoNodoPrimo = new nodoBDD(nombreNodo, this,
    false);
    listaDeNodosPrimos.Add(nombreNodo, nuevoNodoPrimo);
}
```

Figura 4.18: Método CrearNodo

Dentro de la clase *Modelo* programamos todos los métodos necesarios para interactuar con los diagramas binarios de decisión tanto para la construcción inicial del propio diagrama como para su manipulación durante la ejecución del algoritmo de etiquetamiento. Uno de los métodos imprescindibles para utilizar diagramas binarios de decisión junto con el algoritmo de etiquetamiento es la conversión de nodos a sus valores primos y viceversa. De [4] sabemos que la implementación de $check_{EX}$ utiliza la primitiva $pre_{\exists}$ (preimagen existencial). Para utilizar $pre_{\exists}$ con diagramas binarios debemos hacer ajustes al algoritmo de etiquetamiento lo cual implica en algunas situaciones hacer diferencia entre la representación de los nodos primos y los no primos. Discutiremos en la siguiente sección los detalles de los ajustes necesarios para utilizar diagramas binarios de decisión con el algoritmo de etiquetamiento. La implementación de los métodos que nos permiten obtener los valores primos de un conjunto de nodos no primos (Figura 4.19) se consigue mediante intercambios dentro de los diagramas binarios de decisión. Para realizar intercambios dentro de los diagramas primero debemos crear los pares de intercambio, es decir, indicamos el valor “origen” y su valor “destino” antes de llevar a cabo el intercambio. En el caso de *CUDD* es importante destruir los pares de intercambio después de ejecutar la operación; de otra manera no se libera la memoria utilizada.

```
private int obtenerNoPrimos(int conjunto)
{
    //se obtienen las listas ordenadas
    List<nodoBDD> _listaDeNodosOrdenada = listaDeNodosOrdenada();
    List<nodoBDD> _listaDeNodosPrimosOrdenada =
    listaDeNodosPrimosOrdenada();
    int[] destino = new int[_listaDeNodosOrdenada.Count];
```

```
            int[] origen = new int[_listaDeNodosPrimosOrdenada.Count];

            for (int i = 0; i < origen.Length; i++)
            {
                origen[i] = _listaDeNodosPrimosOrdenada[i].EnteroBDD;
                destino[i] = _listaDeNodosOrdenada[i].EnteroBDD;
            }
            //se lleva a cabo la permutación mediante las funciones de
            //JBDD
            int intercambio = bdd.createPair(origen, destino);
            int resultado = bdd.replace(conjunto, intercambio);
            bdd.deletePair(intercambio);
            return resultado;
        }
```

Figura 4.19: Método para obtener valores no primos de un conjunto de nodos primos

En la implementación del método *obtenerNoPrimos* observamos nuevamente el uso del ordenamiento sobre nodos. Las invocaciones a través del objeto *bdd* son invocaciones de funciones en *CUDD* a través de *JBDD*. El método *obtenerPrimos* es equivalente al método *obtenerNoPrimos* con el origen y destino intercambiados.

La ventaja de utilizar diccionarios dentro de la clase *Modelo* para almacenar la lista de nodos es evidente en la implementación del método *leerEnteroDeNodo* (Figura 4.20). La recuperación del entero con el cual *CUDD* representa al nodo se obtiene inmediatamente a través del nombre que nosotros le asignamos.

```
        public int leerEnteroDeNodo(string nombreDeNodo)
        {
            return listaDeNodos[nombreDeNodo].EnteroBDD;
        }
```

Figura 4.20: Método leerEnteroDeNodo

La explotación del diagrama binario de decisión se hace evidente en el algoritmo de etiquetamiento. Para construir el diagrama binario de decisión hacemos uso de la clase *Modelo* que a su vez implica la clase *nodoBDD*. Dentro del verificador de consola tenemos una clase directriz a través de la cual hacemos las diferentes invocaciones dentro del proceso principal de

verificación. La clase directriz *Program* (Figura 4.21) es la clase en la que hacemos la instancia inicial de la clase *Modelo* y en la que corremos el proceso para generar el diagrama binario de decisión. La clase *Program* se compone de un único método *Main*; el método *Main* representa el hilo de ejecución mediante el cual se definen el principio y el fin de la ejecución del verificador de consola. El algoritmo que seguimos para la construcción del diagrama binario de decisión corresponde al mostrado en el capítulo 2.

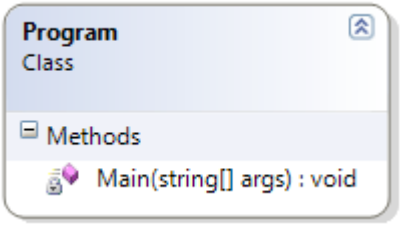


Figura 4.21: Clase Program

La creación del diagrama binario de decisión que representa las tablas de la red genética se puede dividir en pasos. En términos programáticos, el primer paso es la creación del objeto de tipo *Modelo*. El segundo paso es la creación de todos los nodos necesarios para representar las interacciones en el sistema. En este paso únicamente reservamos memoria y ponemos nombre a los nodos. En la Figura 4.22 pueden observarse tanto el paso 1 como el paso 2.

```
Modelo M = new Modelo();

//Crear todos los átomos necesarios a partir de la
//especificación hecha en el archivo
foreach (string nombre in archivoEstructura.ObtenerAtomos())
{
    M.CrearNodo(nombre);
}
```

Figura 4.22: Primer y segundo paso para la creación del diagrama binario de decisión que representa la transición entre nodos

En la Figura 4.22 podemos observar el uso del objeto *archivoEstructura.* Como mencionamos antes, cualquier implementación para lectura de archivos debe hacerse mediante la

implementación de la interfaz *ArchivoEstructura* (Figura 4.7) con lo cual aseguramos que exista el método *ObtenerAtomos*.

El tercer paso para la creación del diagrama binario de decisión es la inicialización del propio diagrama. Para ello hacemos uso de *CUDD* (Figura 4.23).

```
M.BddTransiciones = M.bdd.getZero();
```

Figura 4.23: Tercer paso para la creación del diagrama binario de decisión

Dentro del modelo almacenamos dos diagramas binarios de decisión. Guardamos el diagrama que almacena los estados resultado de la verificación dentro del campo *bddResultado* de la clase *Modelo*, mientras que guardamos el diagrama que almacena las transiciones entre estados del sistema en *bddTransiciones* también de la clase *Modelo*. Durante la ejecución del algoritmo de etiquetamiento en ningún momento modificamos el diagrama que almacena las transiciones.

El cuarto paso es modificar el diagrama binario de decisión que almacena las transiciones dentro del sistema. Para la especificación de las transiciones llevamos a cabo las operaciones descritas en la sección 8.3 del capítulo 2. Podemos observar en la Figura 4.24 el código necesario para construir las transiciones del sistema.

```
foreach (string estadoActual in
         archivoEstructura.ObtenerTransiciones().Keys)
{
    //para definir las transiciones del estado actual se
     //inicia
    //en la constante uno; de esta forma poder realizar la
   //operación 'and'
    int transicionesEstadoActual = M.bdd.getOne();

    //construir el bdd a partir del estado actual
    for (int i = 0; i < estadoActual.Length; i++)
    {
        if (estadoActual[i] == '1')
        {
            transicionesEstadoActual =
              M.bdd.and(transicionesEstadoActual,
```

```
                    M.leerEnteroDeNodo(
                    archivoEstructura.ObtenerAtomos()[i]));
                }
                else if (estadoActual[i] == '0')
                {
                    transicionesEstadoActual =
                     M.bdd.and(transicionesEstadoActual,
                      M.bdd.not(M.leerEnteroDeNodo(
                      archivoEstructura.ObtenerAtomos()[i])));
                }
            }

            //obtener estado primo
            //se obtiene haciendo uso del diccionario con la llave
             // estadoActual
            string estadoSiguiente =
              archivoEstructura.ObtenerTransiciones()[estadoActual];

            //modificar bdd de transiciones para incluir estados
             //primos
            for (int i = 0; i < estadoSiguiente.Length; i++)
            {
                ...
                //Equivalente al ciclo para estado actual; aquí se
                //deben recuperar nodos primos

            }
            M.BddTransiciones = M.bdd.or(M.BddTransiciones,
              transicionesEstadoActual);
        }
```

Figura 4.24: Cuarto paso para la creación del diagrama binario de decisión

En el código de la Figura 4.24 lo que se busca es la construcción de la función que representa la transición:

$$(l_1 \cdot l_2 \cdots l_n) \cdot (l'_1 \cdot l'_2 \cdots l'_n),$$

Es evidente que para construir esta función es necesario el uso del operador *and*. Finalmente deseamos construir la tabla completa que representa todas las transiciones existentes en el sistema; el operador *or* al final del código de la Figura 4.24 nos permite unir todas las funciones de transición en el diagrama binario de decisión *bddTransiciones*.

En este punto hemos leído el archivo de entrada con la especificación en forma de red booleana, y a partir de la lectura de dicho archivo hemos construido el diagrama binario de decisión que guarda (en términos genéticos) todos los genes del sistema y sus relaciones de regulación (*bddTransiciones*). El diagrama binario de decisión que almacena los estados que satisfacen la propiedad verificada (*bddResultado*) en este momento tiene el valor con el que lo inicializamos: la constante 0.

4.4.3 Algoritmo de etiquetamiento extendido

El algoritmo de etiquetamiento que mostramos en esta sección difiere del presentado en forma de pseudo-código en el capítulo 2 en el sentido de que ya integramos la extensión híbrida. Además, agregamos un nuevo operador al cual llamamos *Predecesor*; en el verificador usamos la letra *P.* El operador *P* nos permite obtener los estados sucesores dado un conjunto de estados. Explicaremos los detalles del operador *P* más adelante en esta sección.

El algoritmo de etiquetamiento se implementó dentro de la clase estática *AlgoritmoEtiquetamiento;* en la Figura 4.25 podemos observar el diagrama de la clase *AlgoritmoEtiquetamiento.*

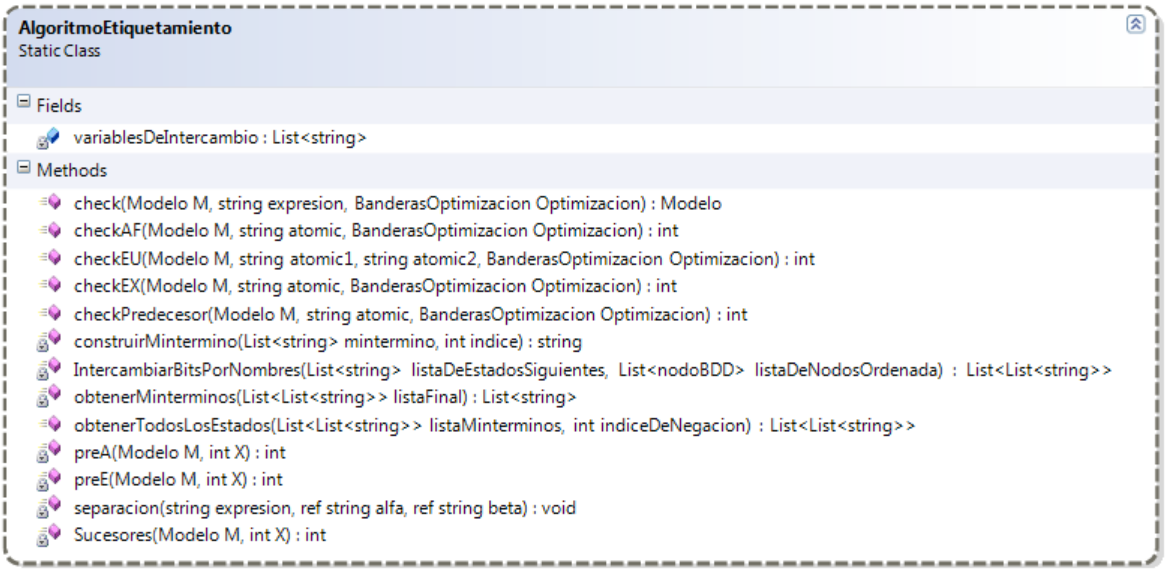


Figura 4.25: Clase estática AlgoritmoEtiquetamiento

La clase *AlgoritmoEtiquetamiento* es estática con la finalidad de no requerir crear objetos para su ejecución. Para hacer uso del algoritmo de etiquetamiento basta con hacer una única invocación al método *check* principal. El método *check* recibe como parámetros de entrada un objeto de tipo *Modelo*, una fórmula *CTL* en forma de cadena que representa la propiedad que deseamos verificar, y también recibe una variable de tipo *BanderasOptimizacion*. El parámetro de tipo *BanderasOptimizacion* no forma parte del algoritmo original presentado en el capítulo 2 y en [4]. En el capítulo 2 de este trabajo mencionamos que la hibridación de la lógica *CTL* aunque resulta en una lógica más expresiva también implica mayor complejidad computacional bajo ciertas circunstancias, en particular mencionamos que el anidamiento de operadores híbridos resulta caro en términos computacionales. Dentro de las propiedades importantes para verificar redes genéticas al menos una se construye mediante anidamiento de operadores híbridos. Resultó importante optimizar dicho anidamiento y de ahí se deriva la necesidad de incluir la variable de tipo *BanderasOptimizacion.*

Implementamos *BanderasOptimizacion* mediante una enumeración; el diagrama de la enumeración *BanderasOptimizacion* lo podemos observar en la Figura 4.26.

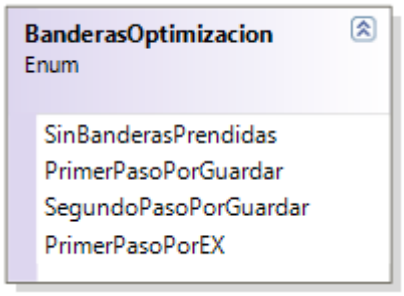


Figura 4.26: Enumeración BanderasOptimizacion

Explicaremos el uso de la enumeración *BanderasOptimizacion* conforme se utilice. En la mayoría de los casos la variable de optimización únicamente se pasará de método en método sin afectar su contenido.

El código que implementa el algoritmo de etiquetamiento es suficientemente extenso como para evitar incluirlo en este texto. Mostraremos el código que creamos más significativo y dejaremos que el lector consulte los demás casos directamente en el código fuente del verificador. Dentro de la implementación del algoritmo hay una serie de métodos auxiliares que nos sirvieron para cumplir con tareas ajenas al propio algoritmo pero necesarias para su implementación.

Como lo mencionamos, el método principal *check* recibe una cadena en la que se especifica la propiedad que deseamos verificar. La cadena *expresion* es resultado del procesamiento léxico y sintáctico que se lleva a cabo en la capa 3 de la arquitectura del sistema. La cadena *expresion* puede ser explotada de forma tal que nos indique los operadores en el orden correcto de ejecución. De esta manera podemos implementar el algoritmo *check* como una toma de decisión que nos permita seleccionar el flujo de ejecución con base en el contenido de la cadena *expresion.* Resulta idóneo utilizar un “switch” en conjunto con el hecho de que una cadena no es más que un arreglo unidimensional de caracteres. En la Figura 4.27 mostramos las primeras líneas del algoritmo *check*.

```
static public Modelo check(Modelo M, string expresion,
                BanderasOptimizacion Optimizacion)
{
    switch (expresion[0])
    {
        case 'T': //top
            M.BddResultado = M.bdd.getOne();
            return M;
        case 'B': //bottom
            M.BddResultado = M.bdd.getZero();
            return M;
```

Figura 4.27: Primeras líneas del algoritmo check

En la Figura 4.27 no se hace uso de la variable de optimización. El uso de la variable *expresion* es únicamente para la selección del caso; el objeto de tipo *Modelo* es importante pues nos permite realizar las operaciones lógicas sobre los diagramas binarios de decisión.

Nuestra implementación del algoritmo *check* regresa como resultado un *Modelo*. En cada caso del código implementado observaremos que al final habrá un retorno de *M* que es un objeto de tipo *Modelo*. Recordemos que almacenamos los estados que satisfacen la propiedad verificada en el diagrama binario de decisión *bddResultado* dentro de la clase *Modelo*. Si observamos el pseudo-código presentado en el capítulo 2 para la implementación de *check,* observaremos que si la fórmula de entrada es *Top,* el conjunto de estados que la satisfacen son todos los estados que conforman el sistema; el caso contrario ocurre cuando la fórmula de entrada es *Bottom*. En términos de verificación híbrida la forma mediante la cual podemos representar a todo el conjunto de estados es mediante la construcción del diagrama binario de decisión que representa la constante 1; para representar el conjunto vacío de estados utilizamos el diagrama binario de decisión que representa la constante 0; podemos observar lo anterior en la Figura 4.27.

La implementación que mostramos en la Figura 4.27 es trivial. Para el resto de los operadores la implementación hace uso en todos los casos de métodos auxiliares o de llamadas recursivas al propio algoritmo *check*.

El resultado del procesamiento de la capa 3 nos arroja expresiones fácilmente procesables. Por ejemplo, podemos asegurar que todos los operadores que lleguen al algoritmo de verificación se

conforman de un único carácter que los identifica y que el alcance del operador siempre se encuentra delimitado por paréntesis; más adelante explicaremos el funcionamiento de la capa 3 del sistema. Para los operadores en *check* que hacen uso de llamadas recursivas debemos primero consumir de la variable *expresion* el propio operador y los paréntesis que lo delimitan. Podemos hacer esto fácilmente mediante la construcción de subcadenas de la variable *expresion* siempre que se trate de operadores unarios; para los operadores binarios necesitamos utilizar el método auxiliar *separacion* el cual nos entrega las dos partes de la fórmula. En la Figura 4.28 podemos observar la implementación del caso para el operador de negación y el caso para el operador conjunción; el primero es unario y el segundo es binario.

```
case '!': //negación
    expresion = expresion.Substring(2); //se elimina
     //primer paréntesis y operador
    expresion = expresion.Substring(0, expresion.Length -
     1); //se elimina último paréntesis
    M.BddResultado = M.bdd.not(check(M, expresion,
       Optimizacion).BddResultado);
    return M;
case '&': //and
    string alfa = "", beta = "";
    separacion(expresion, ref alfa, ref beta);
    int bddAlfa = check(M, alfa,
       Optimizacion).BddResultado;
    int bddBeta = check(M, beta,
       Optimizacion).BddResultado;
    M.BddResultado = M.bdd.and(bddAlfa, bddBeta);
    return M;
```

Figura 4.28: Caso para operador unario y operador binario

El caso de la negación en la Figura 4.28 hace uso de la creación de subcadenas para la eliminación, tanto del propio operador de negación como de los paréntesis que delimitan su alcance. Utilizamos los operadores únicamente para dirigir el flujo de ejecución mientras vamos procesando la variable *expresion*. Podemos observar que para llevar a cabo la negación hacemos uso de la operación *not* implementada sobre los diagramas binarios de decisión.

El caso de la conjunción es más complicado en el sentido de que se trata de un operador binario. Para obtener ambas partes de la fórmula que conforman la conjunción utilizamos paso de variables por referencia. En este caso las variables que pasamos por referencia son la cadena *alfa* y la cadena *beta*. El algoritmo de separación es muy simple: únicamente revisa la cadena *expresion* en busca de los dos límites formados por pares de paréntesis y cada uno lo almacena respectivamente en *alfa* y en *beta*. Una vez que tenemos ambas partes de la conjunción, ejecutamos el algoritmo *check* por separado para *alfa* y para *beta* con lo cual obtenemos dos diagramas binarios; finalmente utilizamos la función *and* sobre diagramas binarios de decisión para obtener el diagrama resultado.

Ejecutamos todas las operaciones booleanas de la fórmula directamente sobre los diagramas binarios de decisión (p. ej. la penúltima línea en la Figura 4.28 ejecuta el operador *and*). Algunos operadores lógicos se pueden resolver a través de equivalencias con otros operadores lógicos, tal es el caso de la implicación. Para los operadores que se pueden resolver mediante fórmulas equivalentes implementamos las equivalencias directamente en el lenguaje intermedio que utilizamos a lo largo del verificador. No existe ningún inconveniente en esto por el hecho de que las equivalencias no cambian y que el lenguaje intermedio es necesario para el uso del verificador. El usuario final no necesita aprender el lenguaje intermedio porque la construcción de las expresiones que utiliza el algoritmo de verificación se realizan automáticamente a partir de fórmulas en una sintaxis similar a las de *CTL* convencional. En el código de la implicación en la Figura 4.29 tenemos un caso que se resuelve mediante su equivalencia con el operador conjunción y con la negación.

```
case '>': // Implicación
    alfa = ""; beta = "";
    separacion(expresion, ref alfa, ref beta);
    M.BddResultado = check(M, "|(!(" + alfa + "))(" +
        beta + ")", Optimizacion).BddResultado;
    return M;
```

Figura 4.29: Caso implicación

Para comenzar la explicación de los operadores temporales primero decidimos mostrar la implementación de las funciones primitivas que conforman el algoritmo. Como se mencionó en el capítulo 2, el algoritmo de etiquetamiento únicamente necesita de un conjunto adecuado de

operadores para su funcionamiento. En nuestro caso el conjunto adecuado de operadores lo conforman AF, EU y EX. El pseudo-código mostrado en la sección 7.1 del capítulo 2 para los operadores así como para las primitivas difiere de la implementación que mostramos en este capítulo por el hecho de que la implementación final incluye uso de diagramas binarios de decisión además de la presencia de la variable de optimización.

En la medida de lo posible decidimos mantener los mismos nombres de variables utilizados en el pseudo-código presentado para el algoritmo de etiquetamiento.

Podemos observar la implementación de la primitiva $pre_{\exists}$ en la Figura 4.30.

```
private static int preE(Modelo M, int X)
{
    //apply(*,B->,Bx')
    int XandTransiciones = M.bdd.and(X, M.BddTransiciones);
    //inicia exist
    //variable temporal para armar los minterminos para exist
    //se obtienen todas las variables del bdd de forma ordenada
    int tmp = M.bdd.getOne();
    foreach (nodoBDD nodo in M.listaDeNodosPrimosOrdenada())
    {
        //se obtiene el valor de cada nodo
        int tmp2 = nodo.EnteroBDD;
        //se construyen minterminos
        tmp = M.bdd.and(tmp, tmp2);
    }
    //exists: bdd y minterminos
    int resultado = M.bdd.exists(XandTransiciones, tmp);
    return resultado;
}
```

Figura 4.30: Primitiva $pre_{\exists}$

En la Figura 4.30 dejamos los comentarios por claridad para el lector. Una vez más notamos que en el manejo de diagramas binarios de decisión mediante *CUDD* no existe como tal la función $apply$; esta se encuentra implícita en los distintos operadores lógicos. En este caso para la implementación de la primitiva $pre_{\exists}$ decidimos dividir en partes lo mostrado en la sección 8.3 del

capítulo 2 ($pre_\exists$ para verificación simbólica). En nuestra implementación suponemos que la variable entera *X*, que es un diagrama binario de decisión (recordemos que *CUDD* hace referencia a los diagramas mediante variable enteras), será tal que represente un conjunto de nodos primos dentro del sistema. Cualquier método que invoque a $pre_\exists$ deberá hacer uso de los métodos auxiliares en *Modelo* para obtener nodos primos de conjuntos de estados.

CUDD requiere la construcción de un "cubo" para poder utilizar el algoritmo $exists$. Un *cubo* es un mintérmino (i.e. conjunción de todas las variables posiblemente negadas); por ejemplo, el cubo para el conjunto de variables $\{v_1, v_2, v_3\}$ es $v1 \,\&\, v2 \,\&\, v3$ [M4]. Para la construcción del cubo se asignan únicamente valores 1 a las variables. La construcción del cubo para la primitiva $pre_\exists$ se puede observar en el ciclo *foreach*. Una vez construido el cubo podemos aplicar la función *exists* de *CUDD*.

La implementación de la primitiva $pre_\forall$ (Figura 4.32) es sencilla debido a que se basa en la implementación de la primitiva $pre_\exists$.

```
private static int preA(Modelo M, int X)
{
    int resultado = M.bdd.not(preE(M, M.bdd.not(X)));
    return resultado;
}
```

Figura 4.32: Primitiva $pre_\forall$

Una vez definidas las primitivas podemos definir la implementación del conjunto adecuado de operadores. Mostramos la implementación del operador *AF* en la Figura 4.33.

```
static public int checkAF(Modelo M, string atomic,
    BanderasOptimizacion Optimizacion)
{
    int X = M.BddTransiciones;
    M = check(M, atomic, Optimizacion);
    int Y = M.BddResultado;
    while (X != Y)
    {
        X = Y;
        Y = M.bdd.or(Y, preA(M,
            M.obtenerValoresPrimosDeConjunto(Y)));
    }   return Y;
}
```

Figura 4.33: Implementación de AF

La implementación del algoritmo para $check_{AF}$ es directa a partir del pseudo-código presentado en el capítulo 2. Considerando que estamos trabajando con diagramas binarios, hacemos la representación de todo el conjunto de estados a través del diagrama de transiciones que fue construido precisamente con el conjunto total de estados definido en el archivo de especificación de entrada. La unión del diagrama binario de decisión *Y* con la pre-imagen absoluta de *Y* la conseguimos a través del operador *or* implementado para diagramas binarios de decisión.

La Figura 4.34 muestra la implementación del operador *EU*.

```
static public int checkEU(Modelo M, string atomic1, string
     atomic2, BanderasOptimizacion Optimizacion)
{
    //X := S
    int X = M.BddTransiciones;

    M = check(M, atomic1, Optimizacion);
    int Z = M.BddResultado;

    M = check(M, atomic2, Optimizacion);
    int Y = M.BddResultado;

    while (X != Y)
    {
        X = Y;
        Y = M.bdd.or(Y, M.bdd.and(Z, preE(M,
           M.obtenerValoresPrimosDeConjunto(Y))));
    }
    return Y;
}
```

Figura 4.34: Implementación de EU

A diferencia del pseudo-código para *EU*, en la implementación que mostramos en la Figura 4.34 decidimos primero obtener $X := S$, únicamente por claridad en el código en C#; el orden con el cual se hacen estas asignaciones es semánticamente irrelevante. En la implementación mantuvimos los nombres para variables presentados en el pseudo-código y para los parámetros

de entrada (las fórmulas) la equivalencia es $\alpha_1 \equiv atomic1$ y $\alpha_2 \equiv atomic2$. Una vez más podemos conseguir la unión y la intersección mostradas en el capítulo 2 para *EU*, mediante los operadores *and* y *or* para diagramas binarios de decisión.

La Figura 4.35 muestra la implementación para el operador *EX.*

```
static public int checkEX(Modelo M, string atomic,
      BanderasOptimizacion Optimizacion)
{
    int X, Y;
    X = check(M, atomic,
        Optimizacion).obtenerBDDResultadoConPrimos();
    Y = preE(M, X);
    return Y;
}
```

Figura 4.35: Implementación de EX

La implementación de $check_{EX}$ es directa; solo hay que considerar que por claridad en el código decidimos separar el anidamiento que presentamos en el pseudo-código.

Teniendo definido el conjunto adecuado de operadores podemos continuar la discusión para un operador que hace uso de ellos. El caso que presentamos en la Figura 4.36 es para cuando la variable *expresion* tiene entre sus operadores *EU*. Sabemos que el operador *EU* es un operador temporal binario. Por lo tanto, hacemos uso del método auxiliar *separacion* y enviamos el flujo de ejecución hacia la implementación de $check_{EU}$.

```
case 'E': //EU
    alfa = ""; beta = "";
    separacion(expresion, ref alfa, ref beta);
    M.BddResultado = checkEU(M, alfa, beta,
      Optimizacion);
    return M;
case 'F': //EF
    expresion = expresion.Substring(2);
    expresion = expresion.Substring(0, expresion.Length -
      1);
    M.BddResultado = check(M, "E(T)(" + expresion + ")",
      Optimizacion).BddResultado;
    return M;
```

Figura 4.36: Caso EU y EF

En la Figura 4.36 también mostramos el caso para el operador temporal *EF. EF* es un operador unario por lo que solo es necesaria la construcción de subcadenas; *EF* se puede obtener a partir de su equivalencia con el operador *EU*; para detalles de la equivalencia consultar el capítulo 2.

Hasta este momento no hemos hecho uso de las banderas de optimización. Antes de explicar la optimización presentamos la fórmula que nos motivó a introducir dicha optimización. Las redes genéticas con las que trabajamos en general tienen una característica supuesta por sus creadores. Se tratan de redes determinadas, es decir, cada nodo del sistema tiene un único sucesor. Como parte de nuestras pruebas de verificación de modelos empleamos una fórmula que nos indicara si en efecto la especificación del sistema era una red determinada. La fórmula que nos permite identificar los estados que tienen un único sucesor incluye el uso de operadores híbridos; la Figura 4.37 muestra la fórmula *CTL* que utilizamos para identificar redes determinadas.

$$\downarrow x.\, EX(\downarrow y.\, @_x AXy)$$

Figura 4.37: Estados que tienen un único sucesor

En la Figura 4.37 mostramos una fórmula con anidamiento de operadores temporales. Sin optimización, obtener los estados que tienen un único sucesor es caro computacionalmente. La razón es que el operador *bind* por omisión en su definición original recorre todos los estados del sistema. Para exponer la implementación, en la Figura 4.38 mostramos la definición de la semántica para el operador *bind*; el resto de la semántica para la hibridación de *CTL* puede consultarse en el capítulo 2.

$M \vDash_s \downarrow x.\alpha$ sii $M \vDash_s \alpha[s/x]$ en donde s/x significa reemplazar todas las ocurrencias de x por s.

Figura 4.38: Semántica para operador bind

Podemos observar que la implementación del operador *bind* implica recorrer todos los estados del sistema en búsqueda de aquel que satisfaga α. Por lo tanto, si consideramos el anidamiento de un *bind* dentro de otro *bind* obtenemos que se recorren todos los estados del sistema por cada estado del sistema. La complejidad es cuadrática en el número de estados. Para sistemas grandes el tiempo de ejecución puede ser intolerable.

Podemos realizar una optimización bajo cierto patrón de anidamiento. En particular la fórmula presentada en la Figura 4.37 tiene el patrón $\downarrow EX \downarrow$, lo que significa que el segundo *bind* debería recorrer *únicamente* los estados que cumplen con EX, es decir los estados sucesores al estado que guarda el primer *bind*. Para la implementación de la optimización encontramos necesario implementar una nueva primitiva. En la Figura 4.39 presentamos la definición de *sucesores*.

$$sucesores(Y) = \{s' \in S | existe\, s, (s \rightarrow s' y\, s \in Y)\}$$

Figura 4.39: Definición de primitiva sucesores

La implementación de la primitiva *sucesores* se puede consultar en el código del verificador y en el apéndice de este trabajo (Figura 7.29); en el código de la implementación fuimos extensos en los comentarios con el fin de facilitar la comprensión del código. Para nuestra optimización bastaba con la primitiva *sucesores*. Sin embargo, hicimos accesible dicha primitiva al usuario con un nuevo operador temporal *Predecesor* (*P*) que permite referirse al pasado inmediato. El operador *P* utiliza la primitiva sucesores debido a que el algoritmo de etiquetamiento recorre el modelo hacia atrás; recordemos que el operador *EX* hace uso de la primitiva pre_{EX}.

La primitiva *sucesores* nos permite obtener de forma eficiente los estados que suceden a un conjunto de estados; de esta manera podemos realizar la optimización del patrón $\downarrow EX \downarrow$ como explicamos a continuación:

- Identificar primer paso por operador $\downarrow$, ejecutar algoritmo normal para *bind*
- Identificar paso por operador EX
- Identificar segundo paso por operador $\downarrow$, ejecutar algoritmo solamente para estados sucesores (usando primitiva *sucesores*) del estado guardado en la primer pasada por $\downarrow$

Si el sistema es determinado, el segundo *bind* se hará con un único estado y la optimización será efectiva. En cambio, en el peor de los casos existe un estado con transiciones hacia todos los estados en el sistema haciendo inefectiva la optimización. Por lo tanto mejora el caso promedio. La identificación del patrón, i.e. primer paso por $\downarrow$, primer paso por EX, segundo paso por $\downarrow$, la conseguimos a través del uso de banderas; en este caso las posibles banderas son las definidas dentro de la enumeración *BanderasOptimizacion*. En la Figura 4.40 mostramos el uso de las banderas en el caso del operador EX.

```
case 'Y': //EX
    if (Optimizacion ==
      BanderasOptimizacion.PrimerPasoPorGuardar)
    {
        Optimizacion =
      BanderasOptimizacion.PrimerPasoPorEX;
    }

    expresion = expresion.Substring(2);
    expresion = expresion.Substring(0, expresion.Length -
      1);
    M.BddResultado = checkEX(M, expresion, Optimizacion);
    return M;
```

Figura 4.40: Caso EX

En la Figura 4.40 observamos que para encender la bandera *PrimerPasoPorEX* se tuvo que pasar primero por el operador $\downarrow$ en cuyo caso se debió prender la bandera *PrimerPasoPorGuardar.* La ejecución de EX no varía por razones de la optimización.

El algoritmo principal del verificador quedó definido en términos del algoritmo de etiquetamiento. Las secciones que siguen son una explicación sobre la implementación de la interfaz gráfica de usuario así como una breve explicación de la conversión al lenguaje intermedio a partir de las fórmulas que se reciben como entrada en el sistema.

4.5 Interfaz gráfica

4.5.1 Diagramas de clase

En esta sección describimos brevemente la implementación de la interfaz gráfica actual. No es nuestro interés ahondar en los detalles de esta capa. Es importante mencionar, no obstante, los puntos principales con el objetivo de permitir su fácil sustitución o mejoramiento.

En la Figura 4.41 mostramos el diagrama de las clases que componen la interfaz gráfica. Algunas clases como *Acerca*, *Imagenes, Resources* y *Settings* no tienen mayor interés pues únicamente nos sirven para mostrar la pantalla de información del verificador, manejar las imágenes de la interfaz

o almacenar la configuración de la interfaz. La implementación de las cuatro clases antes mencionadas es común a cualquier aplicación que corre bajo Windows. Por lo tanto, no haremos ningún tipo de discusión respecto a ellas; el código del verificador es abierto por lo que si se desea se puede revisar el fuente. La clase *Program* únicamente contiene el método principal *Main* a través del cual se construye el objeto de tipo *Form* que representa la ventana de la interfaz.

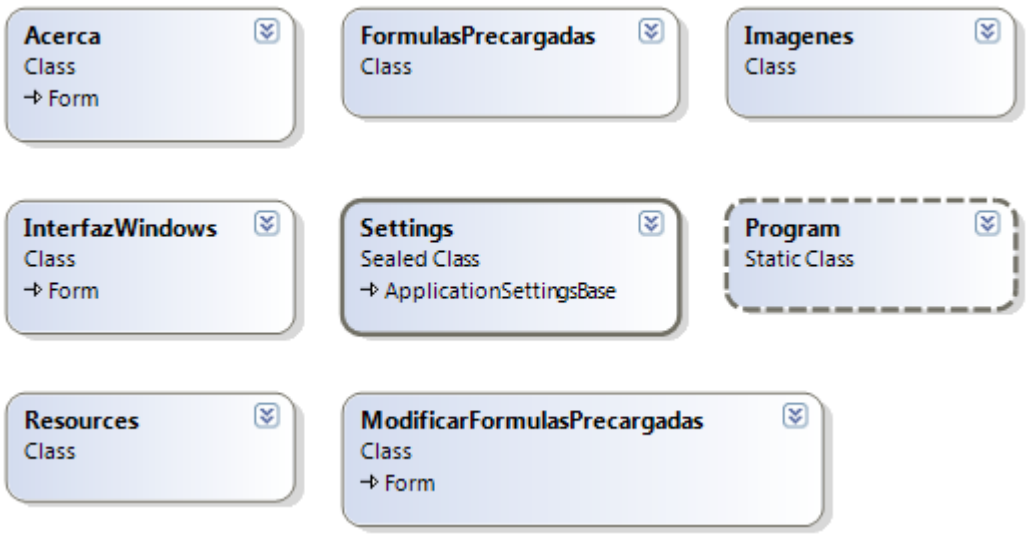


Figura 4.41: Diagramas de clase capa 4

4.5.2 Discusión sobre la implementación

La interfaz gráfica (capa 4 de nuestra arquitectura) se implementó haciendo uso de las librerías contenidas dentro del espacio de nombres *System.Windows.Forms* donde se encuentran los botones y demás controles clásicos de cualquier aplicación de ventanas. Consecuentemente nuestra interfaz gráfica contiene dichos controles. Además se usaron las clases del espacio de nombres *System.Threading* para la interacción con el algoritmo de verificación.

Se implementó la interacción con el usuario en la clase *InterfazWindows*. En la clase *InterfazWindows* se encuentra el código de todos los eventos asociados a los controles de la ventana. La Figura 4.42 muestra el diagrama de la clase *InterfazWindows*.

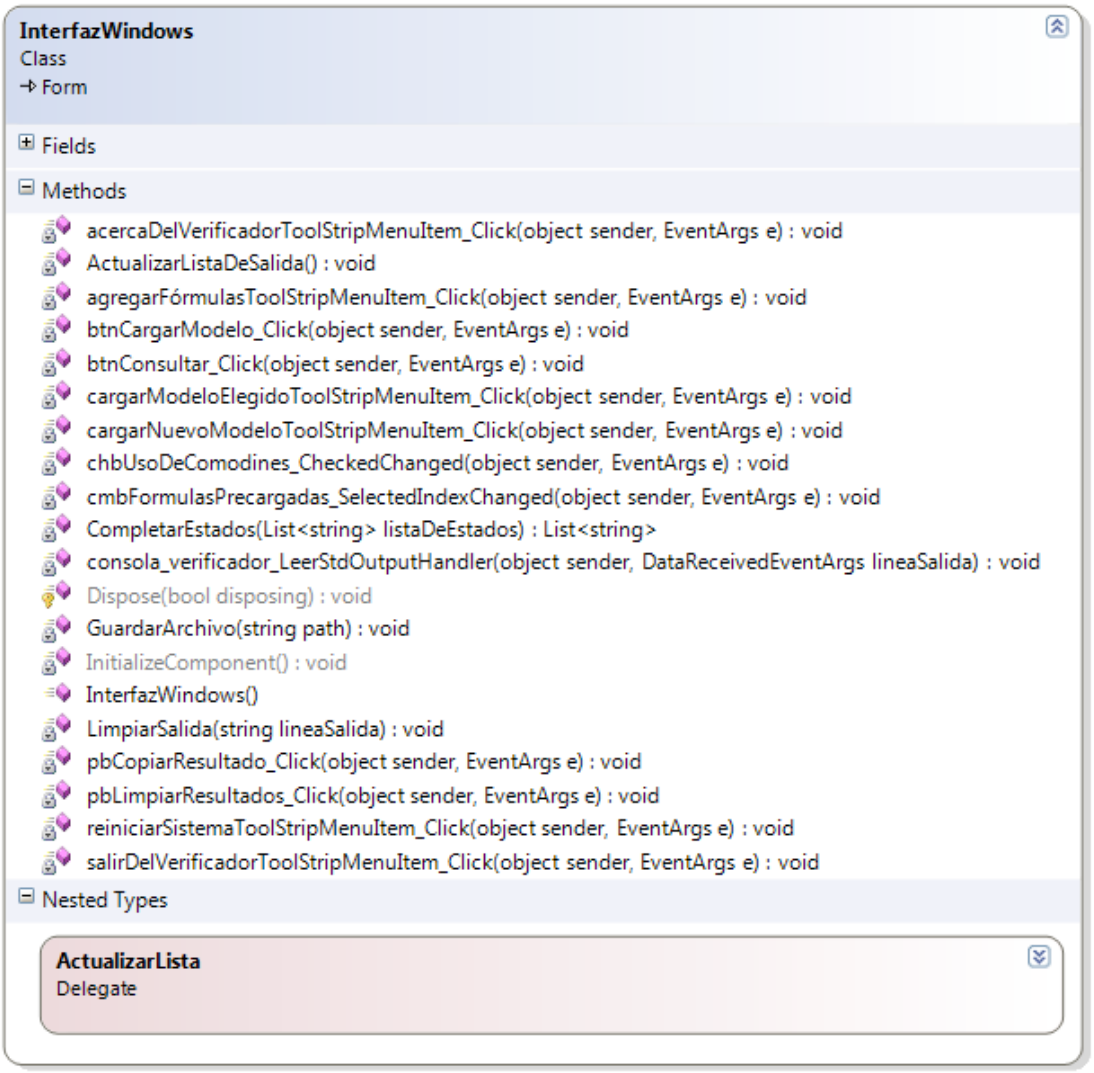


Figura 4.42: Clase InterfazWindows

Por visibilidad ocultamos los campos del diagrama de clases (en total hay 42 campos en la clase *InterfazWindows*). La mayoría de los métodos que se observan en la Figura 4.42 son métodos auxiliares que nos permiten controlar las acciones del usuario sobre la interfaz. La implementación de la interfaz gráfica es interesante por la interacción que lleva a cabo con la capa de verificación; a diferencia del resto de la interfaz, la interacción entre capa 4 y capa 2 del sistema se lleva a cabo de forma asíncrona. La asincronía, que explicaremos a continuación, nos permite liberar el CPU para la ejecución concurrente de otros procesos.

El algoritmo de verificación contenido en la capa 2 de nuestra arquitectura se creó en forma de componente independiente. De esta forma para poder iniciar el componente de verificación es

necesario iniciar un proceso secundario de ejecución. La instancia del proceso de verificación se crea desde la interfaz gráfica mediante el uso de la clase *Process*. Para poder interactuar con el componente de verificación es necesario establecer canales de comunicación; en nuestro caso redireccionamos las salidas estándar del componente de verificación. La comunicación entre el componente de verificación se realiza mediante flujos de escritura y eventos asíncronos de lectura. Si se desea consultar el procedimiento para inicializar el componente de verificación, así como establecer los parámetros de comunicación, se recomienda al lector revisar el código fuente para el evento *Click* del botón *btnCargarModelo*.

La interacción de componentes gráficos con hilos de ejecución asíncronos no es sencilla. Por regla general un proceso *X* no puede tener acceso a objetos creados en un hilo diferente de *X*. En nuestro caso lo anterior implica que no podemos modificar objetos de la interfaz gráfica desde la respuesta asíncrona que se recibe desde el algoritmo de verificación. Esto no es lo que deseamos pues necesitamos presentar los resultados en pantalla. Para poder modificar objetos en procesos ajenos utilizamos el delegado *ActualizarLista* (Figura 4.42). Mediante el delegado *ActualizarLista* igualamos el proceso de ejecución asíncrono mediante el cual recibimos resultados con el proceso de ejecución principal mediante el cual creamos todos los objetos de la interfaz gráfica. Los detalles de esta implementación pueden consultarse en el método *ActualizarListaDeSalida*.

Por facilidad dividimos el código de la interfaz gráfica en regiones. De esta manera es fácil identificar los métodos asíncronos, los eventos, los delegados y los campos de la clase.

En el evento *Click* del botón *btnConsultar* implementamos la creación de los objetos de los analizadores léxico y sintáctico. Antes de enviar la fórmula a verificar procesamos la cadena de entrada de la interfaz gráfica; la fórmula, como lo mencionamos antes, se envía hacia el componente de verificación mediante un flujo de escritura que funciona como canal de comunicación entre el proceso en el cual se ejecuta la interfaz gráfica y el proceso en el cual se ejecuta el verificador.

4.5.3 Interfaz de usuario

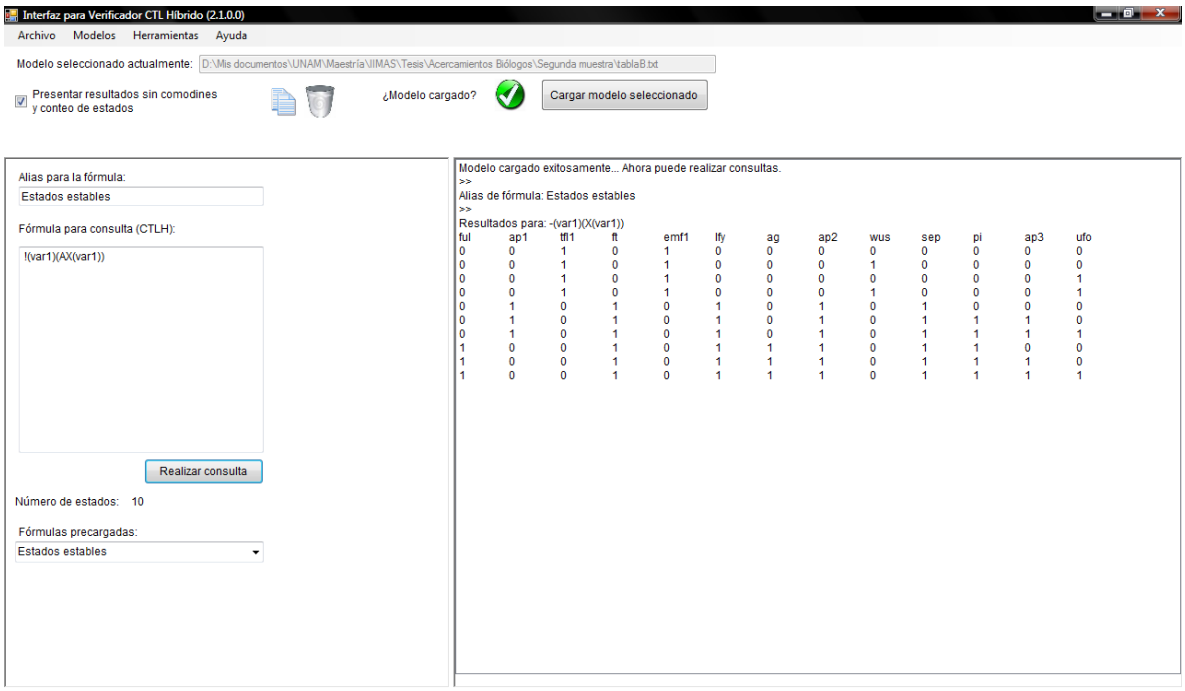


Figura 4.43: Imagen de la interfaz gráfica

En la Figura 4.43 mostramos la interfaz gráfica de nuestro verificador después de verificar cierta propiedad sobre la red genética del primer caso de estudio. En este caso presentamos los resultados sin comodines (observar “checkbox” superior izquierdo) y con conteo de estados (cada renglón representa un estado de la red genética). Como podemos observar la interfaz gráfica es simple. La fórmula que se desea verificar se ingresa en el campo de texto *Fórmula para consulta (CTLH)*, el texto que se ingresa aquí es precisamente el que se envía a procesar para generar la fórmula en el lenguaje intermedio utilizado por la capa 2.

4.6 Analizadores de entrada

4.6.1 Diagramas de clase

La Figura 4.44 presenta el diagrama de las clases que componen nuestro analizador léxico mientras que la Figura 4.45 muestra la única clase que compone el analizador sintáctico.

Figura 4.44: Diagrama de clases analizador léxico

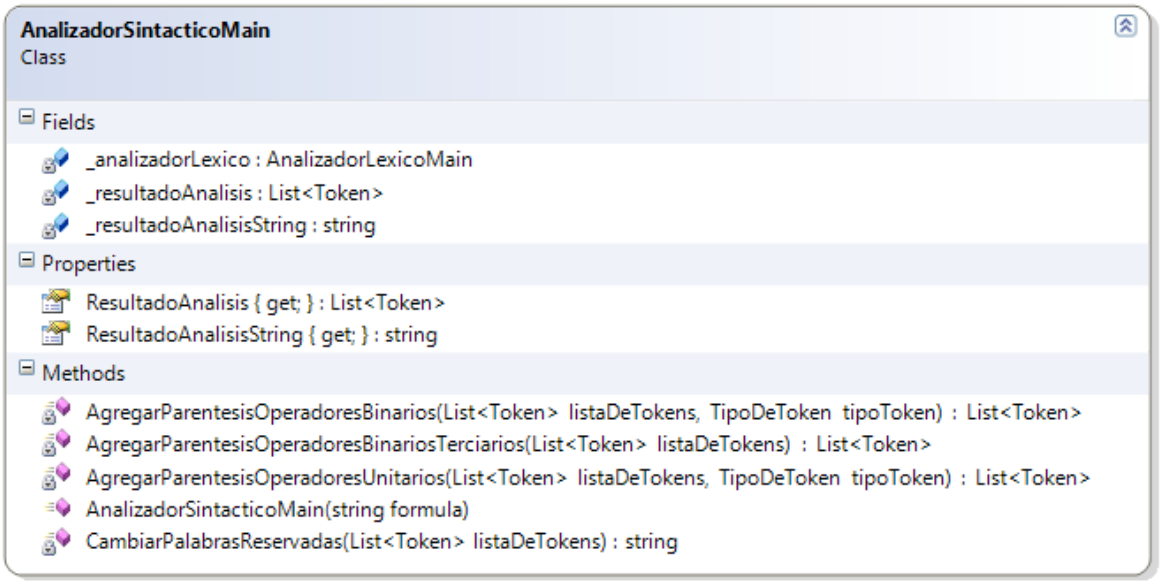


Figura 4.45: Diagramas de clase para analizador sintáctico

La implementación de los analizadores léxico y sintáctico queda fuera de los intereses de este trabajo de tesis; su desarrollo fue simplemente un accesorio para facilitar el uso de nuestro verificador de modelos por parte de las personas del Instituto de Ecología.

Para revisar los detalles de nuestra implementación invitamos al lector a consultar directamente el código fuente de ambos analizadores.

4.6.2 Procesamiento de fórmulas de entrada

Las fórmulas se ingresan al verificador en la sintaxis presentada en el capítulo 2 con la diferencia de que en una computadora es imposible ingresar ciertos símbolos. Para describir el algoritmo para la construcción de las fórmulas en la capa intermedia conviene utilizar un ejemplo.

En la Figura 4.46 mostramos la equivalencia entre los operadores *CTL* y el símbolo que se utiliza en nuestro verificador.

	CTL	*Verificador*	*Capa 2*
1.	T	TOP	T
2.	⊥	BOT	B
3.	¬	NOT	!
4.	∧	AND	&
5.	∨	OR	|
6.	→	IMP	>
7.	AX	AX	X
8.	EX	EX	Y
9.	AU	A(αUδ)	A
10.	EU	E(αUδ)	E
11.	EF	EF	F
12.	EG	EG	G
13.	AF	AF	%
14.	AG	AG	$
15.	$@_s$	@	@
16.	↓x	!	-
17.	∃x	EXISTS	#
18.	P	P	P

Figura 4.46: Equivalencia de símbolos para operadores (α y δ son fórmulas)

Por ejemplo, la siguiente fórmula:

$$AG(p \vee q \vee r \rightarrow EF\ EG\ r)$$

Se ingresaría al sistema así:

```
AG(p OR q OR r IMP EF(EG(r)))
```

Finalmente después del procesamiento, la capa de verificación deberá recibir la fórmula (en lenguaje intermedio):

```
$(>(|(p)(|(q)(r)))(F(G(r))))
```

Los analizadores léxico y sintáctico, además de validar las fórmulas ingresadas, realizan una traducción al lenguaje intermedio. Para poder obtener la fórmula en dicho lenguaje construimos un árbol de sintaxis. Para la fórmula de nuestro ejemplo mostramos el árbol de sintaxis en la Figura 4.47.

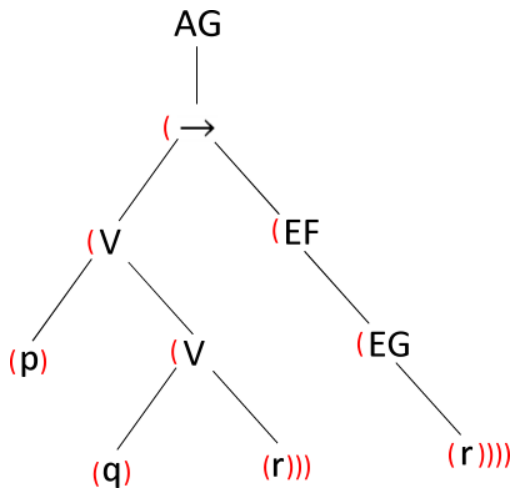


Figura 4.47: Árbol de sintaxis (los paréntesis se agregaron para mostrar el alcance de cada operador)

Entonces la fórmula para el verificador se formaría recorriendo el árbol de sintaxis en preorden sustituyendo los símbolos que sea necesario en base a la Figura 4.46.

La implementación del analizador léxico revisa la sintaxis de la fórmula de entrada y construye las muestras (“tokens” en inglés) necesarias para que la capa del analizador sintáctico construya finalmente el árbol de sintaxis y arroje como salida la fórmula en nuestro lenguaje intermedio.

CAPÍTULO 5

RESULTADOS

5.1 Introducción

En este capítulo presentamos los resultados obtenidos para la flor de la planta *Arabidopsis thaliana*. El resultado que mostramos en el primer caso de estudio para los *atractores de tamaño uno* coincide con los diez estados encontrados en [13]. Comprobamos que la red genética a la cual tuvimos acceso es efectivamente determinada y presentamos algunas fórmulas con resultados para atractores de tamaños mayores a uno.

También mostramos los resultados obtenidos hasta ahora para la red genética de la raíz también de la planta *A. thaliana*.

5.2 Resultados

5.2.1 Primer caso de estudio: flor

5.2.1.1 Atractores de tamaño uno

Fórmula: $\downarrow y.AXy$

Resultados obtenidos con nuestro verificador en Figura 5.1 y Figura 5.2.

ful	ap1	tfl1	ft	emf1	lfy	ag	ap2	wus	sep	pi	ap3	ufo
0	0	1	0	1	0	0	0	*	0	0	0	*
0	1	0	1	0	1	0	1	0	1	0	0	0
0	1	0	1	0	1	0	1	0	1	1	1	*
1	0	0	1	0	1	1	1	0	1	1	0	0
1	0	0	1	0	1	1	1	0	1	1	1	*

Figura 5.1: Resultados $\downarrow y.AXy$ con comodines

ful	ap1	tfl1	ft	emf1	lfy	ag	ap2	wus	sep	pi	ap3	ufo
0	0	1	0	1	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	1	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0	0	1
0	0	1	0	1	0	0	0	1	0	0	0	1
0	1	0	1	0	1	0	1	0	1	0	0	0
0	1	0	1	0	1	0	1	0	1	1	1	0
0	1	0	1	0	1	0	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	0	0
1	0	0	1	0	1	1	1	0	1	1	1	0
1	0	0	1	0	1	1	1	0	1	1	1	1

Figura 5.2: Resultados $\downarrow y.AXy$ sin comodines. 10 estados.

5.2.1.2 Red genética determinada

Fórmula: $\downarrow x.EX(\downarrow y.@_x AX(y))$

Resultados obtenidos con nuestro verificador en Figura 5.3.

ful	ap1	tfl1	ft	emf1	lfy	ag	ap2	wus	sep	pi	ap3	ufo
True												

Figura 5.3: Resultados $\downarrow x.EX(\downarrow y.@_x AX(y))$

5.2.1.3 Atractores de tamaño dos

Fórmula: $\downarrow x.EX(EX(x) \wedge \neg x)$

Resultados obtenidos con nuestro verificador en Figura 5.4.

ful	ap1	tfl1	ft	emf1	lfy	ag	ap2	wus	sep	pi	ap3	ufo
False												

Figura 5.4: Resultados $\downarrow x.EX(EX(x) \wedge \neg x)$

5.2.1.4 Atractores de tamaño tres

Fórmula: $\downarrow x.\, EX(EX(EX(x) \wedge \neg x))$

Resultados obtenidos con nuestro verificador en Figura 5.5.

ful	ap1	tfl1	ft	emf1	lfy	ag	ap2	wus	sep	pi	ap3	ufo
False												

Figura 5.5: Resultados $\downarrow x.\, EX(EX(EX(x) \wedge \neg x))$

5.2.1.5 Atractores de tamaño cuatro

Fórmula: $\downarrow x.\, EX(EX(EX(EX(x) \wedge \neg x)))$

Resultados obtenidos con nuestro verificador en Figura 5.6.

ful	ap1	tfl1	ft	emf1	lfy	ag	ap2	wus	sep	pi	ap3	ufo
False												

Figura 5.6: Resultados $\downarrow x.\, EX(EX(EX(EX(x) \wedge \neg x)))$

5.2.1.6 Cuenca de atracción para primer atractor mostrado en Figura 5.2

Fórmula:

$$EF(\neg ful \wedge \neg ap1 \wedge tfl1 \wedge \neg ft \wedge emf1 \wedge \neg lfy \wedge \neg ag \wedge \neg ap2 \wedge \neg wus \wedge \neg sep \wedge \neg pi \wedge \neg ap3 \wedge \neg ufo)$$

Resultados obtenidos con nuestro verificador en la Figura 5.7.

ful	ap1	tfl1	ft	emf1	lfy	ag	ap2	wus	sep	pi	ap3	ufo
*	0	1	0	1	0	0	*	0	*	*	*	0
*	0	1	0	1	0	1	*	0	*	*	*	0
*	0	1	0	1	0	1	*	1	1	*	*	0
*	0	1	1	1	0	1	*	0	*	*	*	0
*	0	1	1	1	0	1	*	1	1	*	*	0

Figura 5.7: Resultado para cuenca de atracción. En total son 128 estados los que llegan al atractor.

Por razones obvias no presentamos la lista sin comodines.

5.2.1.7 Estados que llegan a alguno de los diez atractores (cuenca de atracción a todos los atractores)

Fórmula: $EF(\downarrow y.AXy)$

Resultados obtenidos con nuestro verificador en la Figura 5.8.

ful	ap1	tfl1	ft	emf1	lfy	ag	ap2	wus	sep	pi	ap3	ufo
True												

Figura 5.8: Resultados $EF(\downarrow y.AXy)$

5.2.2 Segundo caso de estudio: raíz

5.2.2.1 Atractores de tamaño uno

Fórmula: $\downarrow y.AXy$

Resultados obtenidos con nuestro verificador en Figura 5.9 y Figura 5.10.

scr	*shr*	*jkd*	*mgp*	*wox5*	*plt*	*pin*
1	*1*	*1*	*1*	*1*	*	*

Figura 5.9: Resultados $\downarrow y.AXy$ *con comodines*

scr	*shr*	*jkd*	*mgp*	*wox5*	*plt*	*pin*
1	*1*	*1*	*1*	*1*	*0*	*0*
1	*1*	*1*	*1*	*1*	*1*	*0*
1	*1*	*1*	*1*	*1*	*0*	*1*
1	*1*	*1*	*1*	*1*	*1*	*1*

Figura 5.10: Resultados $\downarrow y.AXy$ *sin comodines. 4 estados.*

5.2.2.2 Red genética determinada

Fórmula: $\downarrow x.EX(\downarrow y.@_x AX(y))$

Resultados obtenidos con nuestro verificador en Figura 5.11.

scr	*shr*	*jkd*	*mgp*	*wox5*	*plt*	*pin*
True						

Figura 5.11: Resultados $\downarrow x.\, EX(\downarrow y.\, @_x AX(y))$

5.2.2.3 Atractores de tamaño dos

Fórmula: $\downarrow x.\, EX(EX(x) \wedge \neg x)$

Resultados obtenidos con nuestro verificador en Figura 5.12 y 5.13.

scr	*shr*	*jkd*	*mgp*	*wox5*	*plt*	*pin*
0	*0*	*0*	*0*	*0*	***	***

Figura 5.12: Resultados $\downarrow x.\, EX(EX(x) \wedge \neg x)$ *con comodines*

scr	*shr*	*jkd*	*mgp*	*wox5*	*plt*	*pin*
0	*0*	*0*	*0*	*0*	*0*	*0*
0	*0*	*0*	*0*	*0*	*1*	*0*
0	*0*	*0*	*0*	*0*	*0*	*1*
0	*0*	*0*	*0*	*0*	*1*	*1*

Figura 5.13: Resultados $\downarrow x.\, EX(EX(x) \wedge \neg x)$ *sin comodines. 4 estados.*

5.2.2.4 Atractores de tamaño tres

Fórmula: $\downarrow x.\, EX(EX(EX(x) \wedge \neg x))$

Resultados obtenidos con nuestro verificador en Figura 5.14.

scr	*shr*	*jkd*	*mgp*	*wox5*	*plt*	*pin*
False						

Figura 5.14: Resultados $\downarrow x.\, EX(EX(EX(x) \wedge \neg x))$

5.2.2.5 Atractores de tamaño cuatro

Fórmula: $\downarrow x.\, EX(EX(EX(EX(x) \wedge \neg x)))$

Resultados obtenidos con nuestro verificador en Figura 5.15 y 5.16.

scr	*shr*	*jkd*	*mgp*	*wox5*	*plt*	*pin*
0	*	*0*	*0*	*0*	*	*
1	*0*	*0*	*0*	*1*	*	*

Figura 5.15: Resultados $\downarrow x.\, EX(EX(EX(EX(x) \wedge \neg x)))$ *con comodines*

scr	*shr*	*jkd*	*mgp*	*wox5*	*plt*	*pin*
0	*0*	*0*	*0*	*0*	*0*	*0*
0	*1*	*0*	*0*	*0*	*0*	*0*
0	*0*	*0*	*0*	*0*	*1*	*0*
0	*1*	*0*	*0*	*0*	*1*	*0*
0	*0*	*0*	*0*	*0*	*0*	*1*
0	*1*	*0*	*0*	*0*	*0*	*1*
0	*0*	*0*	*0*	*0*	*1*	*1*
0	*1*	*0*	*0*	*0*	*1*	*1*
1	*0*	*0*	*0*	*1*	*0*	*0*
1	*0*	*0*	*0*	*1*	*1*	*0*
1	*0*	*0*	*0*	*1*	*0*	*1*
1	*0*	*0*	*0*	*1*	*1*	*1*

Figura 5.16: Resultados $\downarrow x.\, EX(EX(EX(EX(x) \wedge \neg x)))$ *sin comodines. 12 estados*

5.2.2.6 Cuenca de atracción para primer atractor mostrado en Figura 5.10

Fórmula: $EF(scr \wedge shr \wedge jkd \wedge mgp \wedge wox5 \wedge \neg plt \wedge \neg pin)$

Resultados obtenidos con nuestro verificador en la Figura 5.17 y 5.18.

scr	*shr*	*jkd*	*mgp*	*wox5*	*plt*	*pin*
1	*1*	*0*	*0*	*	*0*	*0*
1	*1*	*1*	*	*	*0*	*0*

Figura 5.17: Resultado para cuenca de atracción con comodines

scr	shr	jkd	mgp	wox5	plt	pin
1	1	0	0	0	0	0
1	1	0	0	1	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0
1	1	1	0	1	0	0
1	1	1	1	1	0	0

Figura 5.18: Resultado para cuenca de atracción sin comodines. 6 estados.

5.2.2.7 Estados que llegan a alguno de los cuatro atractores (cuenca de atracción a todos los atractores)

Fórmula: $EF(\downarrow y.AXy)$

Resultados obtenidos con nuestro verificador en la Figura 5.19 y 5.20.

scr	*shr*	*jkd*	*mgp*	*wox5*	*plt*	*pin*
1	*1*	*0*	*0*	*	*	*
1	*1*	*1*	*	*	*	*

Figura 5.19: Resultados $EF(\downarrow y.AXy)$ con comodines

scr	*shr*	*jkd*	*mgp*	*wox5*	*plt*	*pin*
1	*1*	*0*	*0*	*0*	*0*	*0*
1	*1*	*0*	*0*	*1*	*0*	*0*
1	*1*	*0*	*0*	*0*	*1*	*0*
1	*1*	*0*	*0*	*1*	*1*	*0*
1	*1*	*0*	*0*	*0*	*0*	*1*
1	*1*	*0*	*0*	*1*	*0*	*1*
1	*1*	*0*	*0*	*0*	*1*	*1*
1	*1*	*0*	*0*	*1*	*1*	*1*
1	*1*	*1*	*0*	*0*	*0*	*0*
1	*1*	*1*	*1*	*0*	*0*	*0*
1	*1*	*1*	*0*	*1*	*0*	*0*
1	*1*	*1*	*1*	*1*	*0*	*0*
1	*1*	*1*	*0*	*0*	*1*	*0*
1	*1*	*1*	*1*	*0*	*1*	*0*
1	*1*	*1*	*0*	*1*	*1*	*0*
1	*1*	*1*	*1*	*1*	*1*	*0*
1	*1*	*1*	*0*	*0*	*0*	*1*
1	*1*	*1*	*1*	*0*	*0*	*1*
1	*1*	*1*	*0*	*1*	*0*	*1*
1	*1*	*1*	*1*	*1*	*0*	*1*
1	*1*	*1*	*0*	*0*	*1*	*1*
1	*1*	*1*	*1*	*0*	*1*	*1*
1	*1*	*1*	*0*	*1*	*1*	*1*
1	*1*	*1*	*1*	*1*	*1*	*1*

Figura 5.20: Resultados $EF(\downarrow y.AXy)$ *sin comodines. 24 estados.*

CAPÍTULO 6
CONCLUSIONES

El grupo de investigación del Laboratorio de Genética Molecular del Instituto de Ecología de la UNAM utiliza para la exploración de sus redes genéticas un simulador. En nuestra opinión el simulador tiene dos desventajas: la primera es inherente al hecho de ser un simulador, y la segunda tiene que ver con la forma en que fue programado. Por un lado, al ser un simulador, se tiene que recorrer cada estado del modelo, i.e. explícitamente se visita cada estado. Por lo tanto, la simulación puede ser un proceso intolerable si el tamaño del modelo es grande. Por otro lado, el simulador se tiene que reprogramar por cada red genética que se desea explorar y también por cada nueva propiedad que se quiere simular. Lo anterior agrega tiempos de espera innecesarios. Al introducir un verificador de modelos eliminamos dichas desventajas. Sin embargo, es posible que existan propiedades que se deseen verificar en biología y que no sean expresables en *CTL* híbrido; en este caso, creemos que la herramienta ideal para el trabajo sería una que combine la verificación de modelos que proponemos con el simulador.

Actualmente, en biología, existen herramientas computacionales que hacen uso de verificación de modelos. Dichas herramientas computacionales permiten a los biólogos analizar grandes cantidades de información para diversos organismos. Entre los muchos organismos para los que se cuenta con información genómica se encuentran: *Homo sapiens*, *Mus musculus* (el ratón común), *Arabidopsis thaliana* (la planta arabidopsis) y *Drosophila melanogaster* (la mosca de la fruta). *Simphatica* y *XSSYS* son dos ejemplos de herramientas computacionales que hacen uso de técnicas de verificación de modelos para estudiar redes celulares y redes bioquímicas [19]. Hoy en día, se ha probado que la verificación de modelos es útil para el análisis de sistemas grandes en el número de estados. La verificación de propiedades complejas con ayuda de la verificación de modelos ha probado ser eficiente. Las técnicas de verificación de modelos no se utilizan únicamente sobre modelos de redes genéticas booleanas, como es nuestro caso. También se ha

demostrado que pueden aplicarse sobre modelos continuos, por ejemplo, modelos de redes genéticas basados en ecuaciones diferenciales [20]. Para nuestro trabajo, las técnicas de verificación de modelos se aplicaron con éxito sobre las redes genéticas definidas por el grupo de investigación del Laboratorio de Genética Molecular.

Con nuestra implementación del verificador de modelos, pudimos reconstruir los diez estados estables (atractores) publicados en [13]; además fuimos capaces de mostrar resultados para otras propiedades de la misma red genética. Con los resultados obtenidos para la flor de la planta *A. thaliana*, comprobamos por un lado los resultados en [13], y por otro lado el funcionamiento de nuestro verificador. Al calcular las cuencas de atracción obtuvimos como resultado conjuntos grandes de estados; en dichos resultados observamos el potencial de la verificación simbólica de modelos. Con ayuda de la verificación de modelos, se aceleró tanto el proceso de análisis de nuevas versiones de redes genéticas, como el análisis de nuevas propiedades. El diseño del modelo matemático de la red genética de la raíz de la planta *A. thaliana* se está realizando con nuestro verificador como herramienta auxiliar.

Una de las contribuciones de nuestro verificador de modelos es la posibilidad de observar el conjunto de estados que satisfacen la propiedad verificada. Como se observó en el capítulo 4, nuestra interfaz presenta en forma de lista cada estado que satisface la fórmula expresada en *CTL*. Otros verificadores no lo permiten, por ejemplo, *NuSMV*.

Hasta donde sabemos, otra contribución de nuestro trabajo es el uso de *CTL* híbrido para calcular conjuntos de estados en redes genéticas booleanas. Al permitir hacer referencia a estados, logramos la especificación de propiedades que con *CTL* simple no es posible. Por ejemplo, calcular los estados atractores de una red genética. Además, mostramos una técnica para optimizar la ejecución de ciertos patrones de anidamiento de operadores híbridos, en particular optimizamos el patrón $\downarrow EX \downarrow$, logrando una mejoría en su caso promedio. Para conseguir dicha optimización, implementamos la primitiva *sucesores* que se aplica directamente sobre los diagramas binarios de decisión. Como consecuencia, hicimos disponible al usuario el operador *P*, que permite hacer referencia al pasado inmediato.

La implementación de la arquitectura por capas cumplió con las propiedades esperadas que se mencionaron en el capítulo 4. La ejecución de pruebas unitarias y pruebas de integración fue sencilla. Los módulos que generamos son fácilmente reutilizables.

En nuestra opinión, el uso de las técnicas de verificación de modelos en biología tiene un gran futuro. La facilidad con la que se incorporaron dichas técnicas en redes genéticas nos hace pensar que su uso puede llegar a ser tan importante como lo es hoy en día para el diseño de sistemas digitales. Enfocamos nuestro verificador a redes genéticas. Sin embargo, el desarrollo de funcionalidades para otro tipo de sistemas es posible.

La implementación de un verificador de modelos implica conocimientos de lógica, de programación y hasta cierto punto, conocimiento del área sobre la cual se verificará; el uso de lenguajes de programación eficientes es obligatorio cuando se desea que el verificador sea utilizado en ambientes reales. En contraparte con la implementación, el uso de un verificador de modelos requiere únicamente conocimientos de lógica y por supuesto, del área de estudio. Con las técnicas de verificación de modelos, nuevas propiedades son fácilmente verificables y el estudio de nuevas redes genéticas es transparente a la implementación. En comparación con el simulador, en el que siempre se requiere que el usuario pueda modificar el código con el cual fue programado.

Nuestro verificador es un primer paso en el desarrollo de un verificador de modelos. Dejamos abiertas varias posibilidades para trabajos futuros. Por ejemplo, la implementación de componentes fuertemente conexas ayudaría a reducir la complejidad del algoritmo de etiquetamiento que tenemos actualmente. También, la representación del modelo de estudio podría mejorarse con el uso de fórmulas en lugar de tablas. En esta primera etapa de trabajo, el desarrollo de los analizadores léxico y sintáctico fue secundario; su implementación puede mejorarse con el uso de herramientas para la construcción de analizadores como lo son *LeX* [M8] y *Yacc* [M9]. Esperamos que nuestro verificador de modelos sea extendido en el futuro por otras personas.

APÉNDICE

Tablas utilizadas en nuestro verificador de modelos para la definición de genes para la flor de la planta Arabidopsis thaliana.

FUL_____AP1 TFL1

00	1
01	0
10	0
11	0

Figura 7.1: Definición de gen FUL

FT_____EMF1

0	1
1	0

Figura 7.2: Definición de gen FT

AP1_____FT LFY AG TFL1

0000	1	1000	1
0001	0	1001	1
0010	0	1010	0
0011	0	1011	0
0100	1	1100	1
0101	1	1101	1
0110	0	1110	0
0111	0	1111	0

Figura 7.3: Definición de gen AP1

EMF1_____LFY

0	1
1	0

Figura 7.4: Definición de gen EMF1

LFY_____FUL AP1 EMF1 TFL1

0000	1	0100	1
0001	1	0101	1
0010	1	0110	1
0011	0	0111	0

1000|1	1100|1
1001|1	1101|1
1010|1	1110|1
1011|0	1111|0

Figura 7.5: Definición de gen LFY

AP2_____TFL1

0|1
1|0

Figura 7.6: Definición de gen AP2

WUS_____WUS AG SEP

000|0
001|0
010|0
011|0
100|1
101|1
110|1
111|0

Figura 7.7: Definición de gen WUS

AG_____AP1 LFY AP2 WUS AG TFL1 SEP

0000000|1	0011001|0	0110010|1
0000001|1	0011010|0	0110011|1
0000010|0	0011011|0	0110100|1
0000011|0	0011100|0	0110101|1
0000100|1	0011101|0	0110110|1
0000101|1	0011110|0	0110111|1
0000110|0	0011111|0	0111000|1
0000111|0	0100000|1	0111001|1
0001000|1	0100001|1	0111010|1
0001001|1	0100010|1	0111011|1
0001010|0	0100011|1	0111100|1
0001011|0	0100100|1	0111101|1
0001100|1	0100101|1	0111110|1
0001101|1	0100110|1	0111111|1
0001110|0	0100111|1	1000000|1
0001111|0	0101000|1	1000001|1
0010000|0	0101001|1	1000010|0
0010001|0	0101010|1	1000011|0
0010010|0	0101011|1	1000100|1
0010011|0	0101100|1	1000101|1
0010100|0	0101101|1	1000110|0
0010101|0	0101110|1	1000111|0
0010110|0	0101111|1	1001000|1
0010111|0	0110000|1	1001001|1
0011000|0	0110001|1	1001010|0

1001011|0	1011101|0	1101111|1
1001100|1	1011110|0	1110000|0
1001101|1	1011111|0	1110001|0
1001110|0	1100000|1	1110010|0
1001111|0	1100001|1	1110011|0
1010000|0	1100010|1	1110100|0
1010001|0	1100011|1	1110101|1
1010010|0	1100100|1	1110110|0
1010011|0	1100101|1	1110111|1
1010100|0	1100110|1	1111000|1
1010101|0	1100111|1	1111001|1
1010110|0	1101000|1	1111010|1
1010111|0	1101001|1	1111011|1
1011000|0	1101010|1	1111100|1
1011001|0	1101011|1	1111101|1
1011010|0	1101100|1	1111110|1
1011011|0	1101101|1	1111111|1
1011100|0	1101110|1	

Figura 7.8: Definición de gen AG

TFL1_____AP1 EMF1 LFY AP2

0000|0	1000|0
0001|0	1001|0
0010|0	1010|0
0011|0	1011|0
0100|1	1100|0
0101|1	1101|0
0110|0	1110|0
0111|0	1111|0

Figura 7.9: Definición de gen TFL1

PI_____AP1 LFY AG PI SEP AP3

000000|0	001101|0	011010|1
000001|0	001110|0	011011|1
000010|0	001111|1	011100|1
000011|0	010000|0	011101|1
000100|0	010001|1	011110|1
000101|0	010010|0	011111|1
000110|0	010011|1	100000|0
000111|0	010100|0	100001|0
001000|0	010101|1	100010|0
001001|0	010110|0	100011|0
001010|0	010111|1	100100|0
001011|0	011000|1	100101|0
001100|0	011001|1	100110|0

100111|1	110000|0	111001|1
101000|0	110001|1	111010|1
101001|0	110010|0	111011|1
101010|0	110011|1	111100|1
101011|0	110100|0	111101|1
101100|0	110101|1	111110|1
101101|0	110110|0	111111|1
101110|0	110111|1	
101111|1	111000|1	

Figura 7.10: Definición de gen PI

SEP_____LFY
0|0
1|1

Figura 7.11: Definición de gen SEP

AP3_____AP1 LFY AG PI SEP AP3 UFO		
0000000|0	0011111|1	0111110|1
0000001|0	0100000|0	0111111|1
0000010|0	0100001|1	1000000|0
0000011|0	0100010|0	1000001|0
0000100|0	0100011|1	1000010|0
0000101|0	0100100|0	1000011|0
0000110|0	0100101|1	1000100|0
0000111|0	0100110|0	1000101|0
0001000|0	0100111|1	1000110|0
0001001|0	0101000|0	1000111|0
0001010|0	0101001|1	1001000|0
0001011|0	0101010|0	1001001|0
0001100|0	0101011|1	1001010|0
0001101|0	0101100|0	1001011|0
0001110|0	0101101|1	1001100|0
0001111|0	0101110|0	1001101|0
0010000|0	0101111|1	1001110|1
0010001|0	0110000|0	1001111|1
0010010|0	0110001|1	1010000|0
0010011|0	0110010|0	1010001|0
0010100|0	0110011|1	1010010|0
0010101|0	0110100|0	1010011|0
0010110|0	0110101|1	1010100|0
0010111|0	0110110|0	1010101|0
0011000|0	0110111|1	1010110|0
0011001|0	0111000|0	1010111|0
0011010|0	0111001|1	1011000|0
0011011|0	0111010|0	1011001|0
0011100|0	0111011|1	1011010|0
0011101|0	0111100|0	1011011|0
0011110|1	0111101|1	1011100|0

1011101|0	1101001|1	1110101|1
1011110|1	1101010|0	1110110|0
1011111|1	1101011|1	1110111|1
1100000|0	1101100|0	1111000|0
1100001|1	1101101|1	1111001|1
1100010|0	1101110|1	1111010|0
1100011|1	1101111|1	1111011|1
1100100|0	1110000|0	1111100|0
1100101|1	1110001|1	1111101|1
1100110|0	1110010|0	1111110|1
1100111|1	1110011|1	1111111|1
1101000|0	1110100|0	

Figura 7.12: Definición de gen AP3

UFO_____UFO
0|0
1|1

Figura 7.13: Definición de gen UFO

Tablas utilizados en nuestro verificador de modelos para la definición de genes para la raíz de la planta Arabidopsis thaliana

SCR____SCR SHR JKD MGP WOX5 PLT PIN	
00***00|0	10***11|0
00***01|0	10***10|0
00***10|0	11*0*01|1
00***11|0	11*0*00|1
01*0*00|1	11*0*11|1
01*0*01|1	11*0*10|1
01*0*10|1	1101*01|0
01*0*11|1	1101*00|0
01*1*00|0	1101*11|0
01*1*01|0	1101*10|0
01*1*10|0	1111*01|1
01*1*11|0	1111*00|1
10***01|0	1111*11|1
10***00|0	1111*10|1

Figura 7.14: Definición de gen SCR

SHR____SCR SHR JKD MGP WOX5 PLT PIN	
00***00|0	01*0*10|0
00***01|0	01*0*11|0
00***10|0	01*1*00|0
00***11|0	01*1*01|0
01*0*00|0	01*1*10|0
01*0*01|0	01*1*11|0

10***01|1 1101*01|1
10***00|1 1101*00|1
10***11|1 1101*11|1
10***10|1 1101*10|1
11*0*01|1 1111*01|1
11*0*00|1 1111*00|1
11*0*11|1 1111*11|1
11*0*10|1 1111*10|1

Figura 7.15: Definición de gen SHR

JKD____SCR SHR JKD MGP WOX5 PLT PIN
00***00|0 10***11|0
00***01|0 10***10|0
00***10|0 11*0*01|1
00***11|0 11*0*00|1
01*0*00|0 11*0*11|1
01*0*01|0 11*0*10|1
01*0*10|0 1101*01|1
01*0*11|0 1101*00|1
01*1*00|0 1101*11|1
01*1*01|0 1101*10|1
01*1*10|0 1111*01|1
01*1*11|0 1111*00|1
10***01|0 1111*11|1
10***00|0 1111*10|1

Figura 7.16: Definición de gen JKD

MGP____SCR SHR JKD MGP WOX5 PLT PIN
00***00|0 10***11|0
00***01|0 10***10|0
00***10|0 11*0*01|1
00***11|0 11*0*00|1
01*0*00|0 11*0*11|1
01*0*01|0 11*0*10|1
01*0*10|0 1101*01|1
01*0*11|0 1101*00|1
01*1*00|0 1101*11|1
01*1*01|0 1101*10|1
01*1*10|0 1111*01|1
01*1*11|0 1111*00|1
10***01|0 1111*11|1
10***00|0 1111*10|1

Figura 7.17: Definición de gen MGP

WOX5____SCR SHR JKD MGP WOX5 PLT PIN
00***00|0 00***11|0
00***01|0 01*0*00|1
00***10|0 01*0*01|1

01*0*10|1	11*0*00|1
01*0*11|1	11*0*11|1
01*1*00|1	11*0*10|1
01*1*01|1	1101*01|1
01*1*10|1	1101*00|1
01*1*11|1	1101*11|1
10***01|0	1101*10|1
10***00|0	1111*01|1
10***11|0	1111*00|1
10***10|0	1111*11|1
11*0*01|1	1111*10|1

Figura 7.18: Definición de gen WOX5

PLT____SCR SHR JKD MGP WOX5 PLT PIN

00***00|0	10***11|1
00***01|0	10***10|1
00***10|1	11*0*01|0
00***11|1	11*0*00|0
01*0*00|0	11*0*11|1
01*0*01|0	11*0*10|1
01*0*10|1	1101*01|0
01*0*11|1	1101*00|0
01*1*00|0	1101*11|1
01*1*01|0	1101*10|1
01*1*10|1	1111*01|0
01*1*11|1	1111*00|0
10***01|0	1111*11|1
10***00|0	1111*10|1

Figura 7.19: Definición de gen PLT

PIN____SCR SHR JKD MGP WOX5 PLT PIN

00***00|1	10***11|1
00***01|0	10***10|0
00***10|1	11*0*01|1
00***11|0	11*0*00|0
01*0*00|1	11*0*11|1
01*0*01|0	11*0*10|0
01*0*10|1	1101*01|1
01*0*11|0	1101*00|0
01*1*00|1	1101*11|1
01*1*01|0	1101*10|0
01*1*10|1	1111*01|1
01*1*11|0	1111*00|0
10***01|1	1111*11|1
10***00|0	1111*10|0

Figura 7.20: Definición de gen PIN

Ejemplos simples de sistemas con sus respectivos archivos de entrada.

Primer ejemplo.

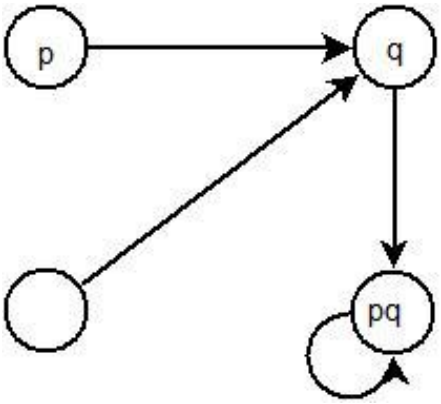


Figura 7.21: Estructura de Kripke con un atractor de tamaño uno

```
p__p q
00|0
01|1
10|0
11|1
q__p q
00|1
01|1
10|1
11|1
```

Figura 7.22: Tabla para Figura 7.21

Segundo ejemplo.

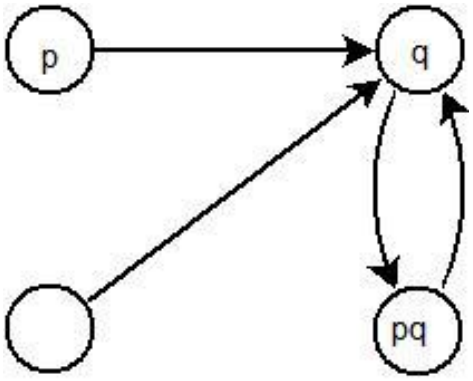


Figura 7.23: Estructura de Kripke con atractores de tamaño dos

```
p__p q
00|0
01|1
10|0
11|0
q__p q
00|1
01|1
10|1
11|1
```

Figura 7.24: Tabla para Figura 7.23

Tercer ejemplo.

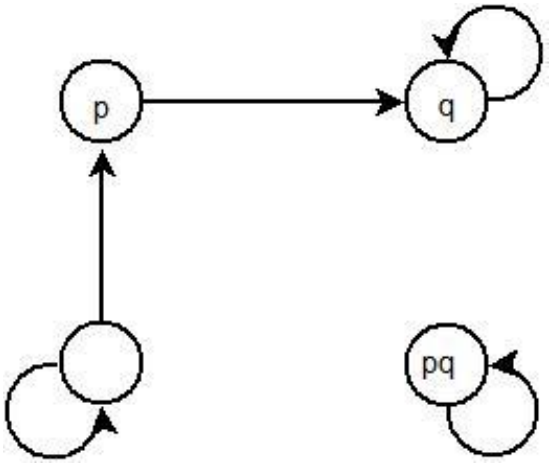


Figura 7.25: Estructura de Kripke con tres atractores de tamaño uno y un estado indefinido

```
p__p q
00|*
01|0
10|0
11|1
q__p q
00|0
01|1
10|1
11|1
```

Figura 7.26: Tabla para la Figura 7.25

Cuarto ejemplo.

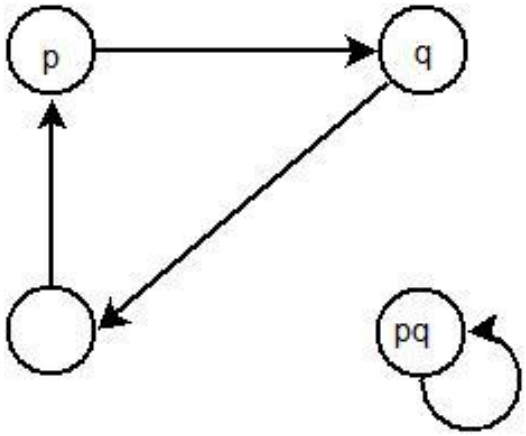


Figura 7.27: Estructura de Kripke con atractores de tamaño tres

```
p__p q
00|1
01|0
10|0
11|1
q__p q
00|0
01|0
10|1
11|1
```

Figura 7.28: Tabla para la Figura 7.27

Comandos necesarios para obtener la DLL necesaria para consumir CUDD en aplicaciones .NET.

- Compilar archivo JAVA con JDK instalado:
  javac JBDD.java
- Incorporar el manifest.mft con el siguiente contenido:
  classpath: .\jbdd.jar
- Crear archivo jar con JDK:
  jar cfm jbdd.jar manifest.mft JBDD.class
- Crear el wrapper con IKVMC:
  ikvmc -target:library jbdd.jar

Implementación de primitiva Sucesores.

```
        private static int Sucesores(Modelo M, int X)
        {
            //apply(*,B->,Bx)
            //XandTransicionesNoPrimos  después de esto contendrá
            //los valores noprimos-primos para la X seleccionada
            int XandTransicionesNoPrimos  = M.bdd.and(X,
M.BddRelaciones);

            //Obtener minterminos con unos (positivo) y con ceros
            //(negativo)
            //sobre las variables no primas
            // ej.   positivo: 1-1-1-1-1-
            //       negativo: 0-0-0-0-0-
            int tmpPositivo = M.bdd.getOne();
            int tmpNegativo = M.bdd.getOne();
            foreach (nodoBDD nodo in M.listaDeNodosOrdenada())
            {

                tmpPositivo = M.bdd.and(tmpPositivo, nodo.EnteroBDD);
                tmpNegativo = M.bdd.and(tmpNegativo,
                                        M.bdd.not(nodo.EnteroBDD));
            }

            // Obtener en un solo termino los minterminos positivos y
             //negativos
            int xor = M.bdd.xor(tmpPositivo, tmpNegativo);
            //negar xor para obtener el resto de los minterminos
            //i.e. quiero tener en xor y en xorNegativo todas las
            //combinaciones de minterminos
            // con únicamente valores sobre los no primos
            // ej. 0-0-      0-1-    1-0-  1-1-
            int xorNegativo = M.bdd.not(xor);

            //intersección entre el nodo con la relación no primo-primo y
            //el valor xor y xorNegativo
            //se obtendrán los valores en xor que cumplen con el estado
            //seleccionado
            int and1 = M.bdd.and(XandTransicionesNoPrimos , xor);
            int and2 = M.bdd.and(XandTransicionesNoPrimos , xorNegativo);

            // Restringir las intersecciones a los valores en X
            // restringir la restricción anterior a los minterminos
            //positivos-negativos
            //i.e. busco quedarme únicamente con las partes primas de
            //XandTransicionesNoPrimos
            //y con no-importa (*) en las partes no-primas
            //para después poder aplicar un replace de variables
            //(permutar primos por no-primos)
            // ++ Se tiene que hacer por separado con cada uno para
            //obtener todas las combinaciones
            // ++  de primos posibles
            int restrict10 = M.bdd.restrict(M.bdd.restrict(and1, X),
                                            tmpNegativo);
```

```
            int restrict11 = M.bdd.restrict(M.bdd.restrict(and1, X),
                                            tmpPositivo);
            int restrict20 = M.bdd.restrict(M.bdd.restrict(and2, X),
                                            tmpNegativo);
            int restrict21 = M.bdd.restrict(M.bdd.restrict(and2, X),
                                            tmpPositivo);

            // Agregar al resultado primer conjunto de estados
            //restringiendo
            //los valores al estado deseado X
            int resultado = M.bdd.restrict(XandTransicionesNoPrimos , X);

            //Si las restricciones positivas-negativas tuvieron alguna
            //respuesta
            //se agregan al resultado
            if (restrict10 != M.bdd.getZero())
            {
                resultado = M.bdd.or(resultado, restrict10);
            }
            if (restrict11 != M.bdd.getZero())
            {
                resultado = M.bdd.or(resultado, restrict11);
            }
            if (restrict20 != M.bdd.getZero())
            {
                resultado = M.bdd.or(resultado, restrict20);
            }
            if (restrict21 != M.bdd.getZero())
            {
                resultado = M.bdd.or(resultado, restrict21);
            }

            //El resultado estará en términos de valores primos
            //se regresa en términos de valores no-primos
            return M.obtenerValoresNoPrimosDeConjunto(resultado);
        }
```

Figura 7.29: Implementación de primitiva Sucesores

GLOSARIO

A

- ***Accesores (C#)***
  En programación orientada a objetos los atributos de una clase deben accederse a través de métodos específicos llamados los accesores.

- ***Ácido Desoxirribonucleico (ADN, DNA en inglés)***
  Contiene toda la información genética usada en el desarrollo y el funcionamiento de los organismos vivos conocidos y de algunos virus, siendo el responsable de su transmisión hereditaria.

- ***Ácido nucleico***
  Los ácidos nucleicos son macromoléculas, polímeros formados por la repetición de monómeros llamados nucleótidos, unidos mediante enlaces fosfodiéster. Se forman, así, largas cadenas o polinucleótidos, lo que hace que algunas de estas moléculas lleguen a alcanzar tamaños gigantes (de millones de nucleótidos de largo).

- ***Ácido ribonucleico (ARN, RNA en inglés)***
  El ácido ribonucleico (ARN) es un ácido nucleico formado por una larga cadena de nucleótidos.

- ***Algoritmo***
  En matemáticas, ciencias de la computación, y disciplinas relacionadas, un algoritmo es una lista bien definida, ordenada y finita de operaciones que permite hallar la solución a un problema. Dado un estado inicial y una entrada, a través de pasos sucesivos y bien definidos se llega a un estado final, obteniendo una solución.

- ***Aminoácido***
  Los aminoácidos son los monómeros de las proteínas.

- ***Árbol (estructura de datos)***
  En ciencias de la computación, un árbol es una estructura de datos ampliamente usada que imita la forma de un árbol (un conjunto de nodos conectados). Un nodo es la unidad sobre la que se construye el árbol y puede tener cero o más nodos hijos conectados a él. Se dice que un nodo a es padre de un nodo b si existe un enlace desde a hasta b (en ese caso, también decimos que b es hijo de a). Sólo puede haber un único nodo sin padres, que llamaremos raíz. Un nodo que no tiene hijos se conoce como hoja. Los demás nodos (tienen padre y uno o varios hijos) se les conoce como rama.

- ***Árbol de ejecución***
  Árbol infinito que representa todas las posibles ejecuciones que pueden generarse a partir de una estructura de Kripke.

- ***Arista (gráficas)***
  En teoría de gráficas las aristas, junto con los vértices, forman los elementos principales con los que trabaja esta disciplina, siendo consideradas las aristas las uniones entre nodos o vértices.

- ***Atributo (programación orientada a objetos)***
  Valor concreto (dato) que forma parte de la especificación del estado de un objeto creado a partir de la definición de una clase.

B

- ***Backus Naur Form (BNF)***
  El Backus-Naur form (BNF) (también conocido como Backus-Naur formalism, Backus normal form o Panini-Backus Form) es una metasintaxis usada para expresar gramáticas libres de contexto: es decir, una manera formal de describir lenguajes formales.

C

- ***C#***
  C# (pronunciado "si sharp") es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET, que después fue aprobado como un estándar por la ECMA e ISO.
  Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma.NET el cual es similar al de Java aunque incluye mejoras derivadas de otros lenguajes (entre ellos Delphi).

- ***Campo (en C#)***
  Ver Atributo.

- ***Célula (biología)***
  Una célula (del latín cellula, diminutivo de cella, hueco) es la unidad morfológica y funcional de todo ser vivo. De hecho, la célula es el elemento de menor tamaño que puede considerarse vivo.

- ***Clase (programación orientada a objetos)***
  Las clases son declaraciones o abstracciones de objetos, lo que significa, que una clase es la definición de un objeto. Cuando se programa un objeto y se definen sus características y funcionalidades, realmente se programa una clase.

- ***Clase abstracta (programación orientada a objetos)***
  Una clase abstracta es una clase con la finalidad de ser heredada. No pueden instanciarse objetos a partir de una clase abstracta.

- ***Clase estática (programación orientada a objetos)***
  Clase a la cual se tiene acceso (a atributos y métodos) sin la necesidad de la instanciación de un objeto.

D

- ***Debugging***
  Depuración. Detección, localización y eliminación de errores en un programa.

- ***Delegado (en C#)***
  Permite redirigir el flujo de una acción. Delega la ejecución hacia un método definido.

E

- ***Encapsulamiento (programación orientada a objetos)***
  En programación modular, y más específicamente en programación orientada a objetos, se denomina encapsulamiento al ocultamiento del estado, es decir, de los datos miembro o atributos, de un objeto de manera que sólo se puede cambiar mediante las operaciones definidas para ese objeto.

- ***Enlace covalente***
  En química, las reacciones entre dos átomos no metales producen enlaces covalentes.

- ***Enlace químico***
  Un enlace químico es la unión entre dos o más átomos para formar una entidad de orden superior, p.ej. una molécula.

- ***Enzima***

Catalizador biológico, normalmente una proteína, que media y promueve un proceso químico sin ser ella misma alterada o destruida.

- ***Escalabilidad (en software)***
  Capacidad de un software o de un hardware de crecer, adaptándose a nuevos requisitos conforme cambian las necesidades del negocio.

G

- ***Gen***
  Secuencia de ácido desoxirribonucleico (ADN) que constituye la unidad funcional para la transmisión de los caracteres hereditarios.

- ***Genéricos (en C#)***
  Mecanismo mediante el cual C# con una sola pieza de código (método, clase, etc.) puede manipular distintos tipos de dato.

- ***Genoma***
  El genoma es todo el material genético contenido en las células de un organismo en particular.

- ***Get (accesor, C#)***
  Propiedad que permite la lectura de un atributo de clase de forma segura.

- ***Glúcido***
  Los glúcidos, carbohidratos o sacáridos (del griego σάκχαρον que significa "azúcar") son moléculas orgánicas compuestas por Carbono, Hidrógeno y Oxígeno. Son solubles en agua y se clasifican de acuerdo a la cantidad de carbonos o por el grupo funcional que tienen adherido. Son la forma biológica primaria de almacenamiento y consumo de energía.

- ***Gramática libre de contexto***
  En lingüística y computación, una gramática libre de contexto es una gramática formal en la que cada regla de producción es de la forma:
  $V \to w$
  Donde V es un símbolo no terminal y w es una cadena de terminales y/o no terminales. El término libre de contexto se refiere al hecho de que el no terminal V puede siempre ser sustituido por w sin tener en cuenta el contexto en el que ocurra. Un lenguaje formal es libre de contexto si hay una gramática libre de contexto que lo genera.

H

- ***Hardware***
  Dispositivos físicos que comprenden un sistema de computación.

- ***Herencia***
  Es una propiedad que permite que los objetos sean creados a partir de otros ya existentes, obteniendo características (métodos y atributos) similares a los ya existentes. Es la relación entre una clase general y otra clase más específica. Es un mecanismo que nos permite crear clases derivadas a partir de clase base, Nos permite compartir automáticamente métodos y datos entre clases subclases y objetos.

I

- ***Interfaz (programación orientada a objetos, en C#)***
  Abstracción de los prototipos de métodos. En C# una interfaz no puede definir prototipos para atributos.

J

- ***JAVA***
  Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria.

L

- ***Lenguaje formal***
  En matemáticas, lógica, y ciencias de la computación, un lenguaje formal es un conjunto de palabras (cadenas de caracteres) de longitud finita en los casos más simples o expresiones válidas (formuladas por palabras) formadas a partir de un alfabeto (conjunto de caracteres) finito.

M

- ***Macromolécula***
  Las macromoléculas son moléculas que tienen una masa molecular elevada, formadas por un gran número de átomos. Generalmente se pueden describir como la repetición de una o unas pocas unidades mínimas o monómeros, formando los polímeros.

- ***Mantenibilidad (en software)***
  Propiedad deseable de un sistema que representa la cantidad de esfuerzo requerida para conservar su funcionamiento normal.

- ***Método (programación orientada a objetos)***
  Representa las acciones que puede llevar a cabo un objeto. Mecanismo mediante el cual se comunican los objetos.

- ***Molécula***
  Una molécula es una partícula formada por un conjunto de átomos ligados por enlaces covalentes o metálicos.

- ***Monómero***
  El monómero (del griego mono, uno y meros, parte) es una molécula de pequeña masa molecular que unida a otros monómeros, a veces cientos o miles, por medio de enlaces químicos, generalmente covalentes, forman macromoléculas llamadas polímeros. Además son unidades básicas o moléculas orgánicas relativamente simples, con estructura definida, estabilizada y específica. Algunos monómeros: Monosacáridos, Ácidos Grasos, Nucleótidos, Aminoácidos, etc.

- ***Monosacárido***
  Los monosacáridos o azúcares simples son los glúcidos más sencillos, que no se hidrolizan, es decir, que no se descomponen para dar otros compuestos,

N

- ***.NET***
  .NET es un proyecto de Microsoft para crear una nueva plataforma de desarrollo de software con énfasis en transparencia de redes, con independencia de plataforma de hardware y que permita un rápido desarrollo de aplicaciones. Basado en ella, la empresa intenta desarrollar una estrategia horizontal que integre todos sus productos, desde el sistema operativo hasta las herramientas de mercado.

- ***Nucleótido***
  Los nucleótidos son moléculas orgánicas formadas por la unión covalente de un monosacárido de cinco carbonos (pentosa), una base nitrogenada y un grupo fosfato.

O

- ***Objeto (programación orientada a objetos)***
  En el paradigma de programación orientada a objetos (POO, o OOP en inglés), un objeto se define como la unidad que en tiempo de ejecución realiza las tareas de un programa. También a un nivel más básico se define como la instancia de una clase.

P

- ***Polímero***
  Los polímeros son macromoléculas (generalmente orgánicas) formadas por la unión de moléculas más pequeñas llamadas monómeros.

- ***Polimorfismo (programación orientada a objetos)***
  En programación orientada a objetos se denomina polimorfismo a la capacidad que tienen los objetos de una clase de responder al mismo mensaje o evento en función de los parámetros utilizados durante su invocación. Un objeto polimórfico es una entidad que puede contener valores de diferentes tipos durante la ejecución del programa.

- ***Programación orientada a objetos (POO)***
  La Programación Orientada a Objetos (POO u OOP según sus siglas en inglés) es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas de computadora. Está basado en varias técnicas, incluyendo herencia, modularidad, polimorfismo y encapsulamiento. Su uso se popularizó a principios de la década de 1990.

- ***Propiedad (en C#)***
  Ver Accesores.

- ***Proteína***
  Las proteínas son macromoléculas formadas por cadenas lineales de aminoácidos.

R

- ***Reusabilidad (en software)***
  Propiedad deseable de un sistema computacional; permite la reutilización de código evitando así redundancias.

- ***Robustez (en software)***
  Propiedad deseable de un sistema computacional; tiene que ver con la calidad del sistema; mientras un sistema tiene mayor robustez se asegura la obtención de los resultados esperados aún en condiciones no conocidas.

S

- ***Seguridad (en software)***
  Propiedad deseable de un sistema computacional; permite asegurar la ejecución de software sin poner en peligro componentes adicionales, además de suponer un uso eficiente de los recursos del sistema.

- ***Set (accesor)***
  Propiedad que permite la escritura de un atributo de clase de forma segura.

- ***Software***
  Conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora.

T

- ***Texto plano (archivo)***
  Los archivos de texto plano (en inglés plain text) son aquellos que están compuestos únicamente por texto sin formato, sólo caracteres.

V

- Vértice (gráficas)
  En teoría de gráficas, un vértice o nodo es la unidad fundamental de la que están formadas las gráficas. Una gráfica no dirigida está formada por un conjunto de vértices y un conjunto de aristas (pares no ordenados de vértices), mientras que una gráfica dirigida está compuesta por un conjunto de vértices y un conjunto de arcos (pares ordenados de vértices). En este contexto, los vértices son tratados como objetos indivisibles y sin propiedades, aunque puedan tener una estructura adicional dependiendo de la aplicación para la cual se usa la gráfica.

Bibliografía y mesografía

Bibliografía

[1] Clarke, E., Gupta, A., Jain, H., Veith, H., Model Checking: Back and Forth Between Hardware and Software, *Lecture Notes in Computer Science*, Vol. 4171/2008, pp. 251–255, Springer, 2008.

[2] Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., Dill, D.L., Symbolic Model Checking for Sequential Circuit Verification, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, pp. 401–424, 1994.

[3] Ben-Ari, M., Manna, Z., and Pnueli, A., The Temporal Logic of Branching Time. *8th Annual ACM Symposium on Principles of Programming Languages*, pp. 164–176, 1981.

[4] Huth, M., Ryan, M., *Logic in Computer Science Modelling and Reasoning about Systems*, Cambridge University Press, 2004.

[5] Clarke, E., Emerson, E.A., Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic*, Lecture Notes in Computer Science*, Vol. 131/1982, pp. 52–71, Springer, 1982.

[6] Franceschet, M., de Rijke, M., M., Model Checking Hybrid Logics (with an Application to Semistructured Data), *Journal of Applied Logic*, Vol. 4, No. 2, pp. 168–191, Elsevier Science, 2006.

[7] Areces, C., ten Cate, B., Hybrid Logics, *Handbook of Modal Logics*, pp. 821–868, Elsevier, 2006.

[8] Vardi, M., Branching vs. Linear Time: Final Showdown, *Lecture Notes in Computer Science*, Vol. 2031/2001, pp. 1–22, Springer, 2001.

[9] Burch, J., Clarke, E., McMillan, K., Symbolic Model Checking: 10^{20} States and Beyond, *Logic in Computer* Science, LICS '90, Proceedings., pp. 428–439, Fifth Annual IEEE Symposium, 1990.

[10] de Jong, H., Modeling and Simulation of Genetic Regulatory Systems: A Literature Review, *Journal of Computational Biology*, Vol. 9, No. 1, pp. 67–103, 2002.

[11] Shmulevich, I., Dougherty, E., Zhang, W., From Boolean to Probabilistic Boolean Networks and Models of Genetic Regulatory Networks, *Proceedings of the IEEE*, Vol. 90, No. 11, pp. 1778–1792, 2002.

[12] Espinosa-Soto, C., Padilla-Longoria, P., Alvarez-Buylla, E., A Gene Regulatory Network Model for Cell-Fate Determination during Arabidopsis thaliana Flower Development That Is Robust and Recovers Experimental Gene Expression Profiles*, The Plant Cell*, Vol. 16, pp. 2923–2939, 2004.

[13] Chaos, A., Aldana, M., Espinosa-Soto, C., García, B., Garay, A., Alvarez-Buylla, E., From Genes to Flower Patterns and Evolution: Dynamic Models of Gene Regulatory Networks*, Journal of Plant Grouth Regulation*, Springer, pp. 278–289, 2006.

[14] Booch, G., *El Lenguaje Unificado de Modelado*, Ed. Pearson, 2004.

[15] Marshal, D., *Programming Microsoft Visual C# 2005: The language*, Microsoft Press, 2006.

[16] Howard, M., *Writing secure code*, Microsoft Press, 2003.

[17] McConnell, S., *Code complete*, 2da. Edición*,* Microsoft Press, 2004.

[18] Sharp, J., *Microsoft Visual C# 2005 Step by Step*, Microsoft Press, 2005.

[19] Antoniotti, M., Policriti, A., Ugel, N., Mishra, B., Model Building and Model Checking for Biochemical Processes, *Cell Biochemistry and Biophysics*, Vol. 38, Humana Press Inc., pp. 271–286, 2003.

[20] Batt, G., Ropers, D., de Jong, H., Geiselmann, J., Mateescu, R., Page, M., Schneider, D., Analysis and Verification of Qualitative Models of Genetic Regulatory Networks: A Model-Checking Approach, *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, in L. P. Kaelbling and A. Saffiotti, eds., pp. 370–375, 2005.

[21] Chabrier-Rivier, N., Chiaverini, M., Danos, V., Fages, F., Schächter, V., Modeling and querying biomolecular interaction networks, *Theoretical Computer Science*, pp. 25–44, Elsevier, 2004.

Mesografía

[M1] Hybrid Logics Web Page, http://hylo.loria.fr, 7 de enero de 2009.

[M2] KEGG – Current Statistics, http://www.genome.jp/kegg/docs/statistics.html, 13 de enero de 2009.

[M3] CUDD: CU Decision Diagram, http://vlsi.colorado.edu/~fabio/CUDD/, 28 de enero de 2009.

[M4] Arash's BDD page, http://javaddlib.sourceforge.net/jbdd/, 28 de enero de 2009.

[M5] SourceForge.net:buddy, http://sourceforge.net/projects/buddy/, 28 de enero de 2009.

[M6] The JDD project, http://javaddlib.sourceforge.net/jdd/, 28 de enero de 2009.

[M7] IKVM.NET Home Page, http://www.ikvm.net/, 29 de enero de 2009.

[M8] Herramienta de programación lex - Wikipedia, http://es.wikipedia.org/wiki/Herramienta_de_programaci%C3%B3n_lex, 12 de marzo de 2009.

[M9] Yacc – Wikipedia, http://es.wikipedia.org/wiki/Yacc, 12 de marzo de 2009.