UNIVERSIDAD NACIONAL DE RIO CUARTO

Trabajo de tesis para la obtención del grado de Licenciatura en Ciencias de la Computación

Título : Verificación de modelos con Cálculo-μ

Autor
Luciano Putruele

Director de Tesis: *Dr. Pablo F. Castro*
Co-Director de Tesis: *Dr. Germán Regis*

Rio Cuarto, Argentina
Abril 2016

Resumen

En esta tesis desarrollaremos una herramienta de verificación de modelos (Model Checking), llamada MC2, sobre modelos de sistemas formalizados en un lenguaje de descripción simple que también será desarrollado en este trabajo. La herramienta de verificación se encargará de, valga la redundancia, verificar propiedades, caracterizadas en la forma de la lógica temporal Cálculo-μ (o cálculo-Mu), sobre algún modelo descripto. Cabe destacar que tanto el lenguaje de descripción de modelos como el verificador forman parte de la misma herramienta, incluso las propiedades a verificar se deben especificar en la descripción.

Adicionalmente, los descripciones MC2 se basan en la definición de reglas de transición usando proposiciones lógicas atómicas con lo cual es transparente ver la estructura del modelo como una máquina de transición de estados más allá de que internamente se los trata simbólicamente como fórmulas para mejorar el rendimiento.

A todo esto, el metalenguaje utilizado para el desarrollo de estas herramientas es Haskell. Esto también significó una motivación para el desarrollo de esta tesis ya que la utilización de un lenguaje funcional como Haskell para el desarrollo de la herramienta en su totalidad, en lugar de la utilización lenguajes imperativos u orientados a objetos, da un enfoque distinto al comportamiento de la herramienta.

Lo que se propone con este proyecto es explorar otras alternativas para especificar propiedades que un modelo deba satisfacer, siendo la alternativa en este trabajo el Cálculo Mu, en vez de las lógicas temporales más utilizadas en este tipo de herramientas, como ser LTL, CTL y CTL*, ya que esto trae la ventaja de que el Cálculo-Mu es más expresivo que los anteriormente nombrados. MC2 puede servir como herramienta de bajo nivel, en el sentido de que se puede extraer una descripción MC2 a partir de modelos de mas alto nivel, y asi mismo, propiedades en lógicas temporales de más alto nivel, para asi despues usar el verificador MC2 sobre la descripción extraida.

Agradecimientos

Agradezco a la universidad, el departamento de computación, y en especial a mi familia y mis amigos, esto no seria posible sin ellos.

Lista de figuras

Contenidos

Capítulo 1

Introducción

La verificación de modelos o comunmente *model checking* es una técnica automática para verificar sistemas reactivos con una cantidad finita de estados, por ejemplo protocolos de comunicación y diseños de circuitos. Las especificaciones de las propiedades a verificar son expresadas en una lógica temporal proposicional, y el sistema esta modelado como un grafo. Se utiliza una busqueda eficiente para determinar automáticamente si las especificaciones son satisfechas por el grafo [1]. Esta técnica fue desarrollada originalmente en 1981 por Clarke y Emerson. Quielle y Sifakis descubrieron independientemente una técnica similar de verificación poco después. Esta técnica tiene varias ventajas importantes sobre probadores de teoremas para verificación de circuitos y protocolos. La mas importante es que es automática. Normalmente, el usuario provee una representación de alto nivel del modelo y una especificación de la propiedad que se desea verificar. El model checker terminará devolviendo la respuesta $True$ indicando que el modelo satisface la especificación o dará una traza de ejecución a modo de contraejemplo si el modelo no satisface la propiedad. Esta es una propiedad muy importante a la hora de encontrar bugs sutiles.

1.1 Verificación de modelos

1.2 Objetivos

El objetivo principal de este proyecto es explorar otras alternativas para especificar propiedades que un modelo deba satisfacer, siendo la alternativa en este trabajo el cálculo-μ, en lugar de las lógicas temporales más utilizadas en este tipo de herramientas, como ser LTL, CTL y CTL*, además de que esto trae la ventaja de que el Cálculo-μ es más expresivo que los anteriormente nombrados, por lo tanto, las propiedades descritas en LTL,CTL y CTL* pueden ser descritas también usando cálculo-μ. Como objetivo secundario cabe destacar la utilización del paradigma funcional de programación para el desarrollo de la herramienta en su totalidad, en lugar de utilizar paradigmas imperativos u orientados a objetos que son normalmente mas utilizados en el área. En cuanto a la aplicación práctica de la herramienta, la misma esta planeada para verificar propiedades en lógicas donde el problema de la verificación de modelos pueda ser reducida a cálculo-mu, por ejemplo dCTL [2], una lógica temporal deóntica usada para especificar propiedades sobre sistemas tolerantes a fallos.

1.3 Estructura

En el capítulo 2 analizaremos conceptos básicos para la comprensión de esta tesis, conceptos como la verificación de modelos, representación de estos modelos, el concepto de lógicas temporales, y en particular, el cálculo-μ. En el capítulo 3 veremos formas de representar los datos de una manera mas concisa e introduciremos la noción de verificación simbólica de modelos. En el capítulo 4 nos adentramos en la herramienta MC2, su lenguaje de descripción de modelos y su verificador de modelos. Se entra en detalle con la sintaxis y la semántica del lenguaje. Para terminar, se analizarán cuestiones de diseño e implementación de la herramienta. En el capítulo 5 aplicaremos la herramienta a algunos ejemplos concretos para afianzar el entendimiento de la aplicación práctica de esta herramienta.

1.4 Desarrollo

Primero se desarrollo una versión del verificador con estados explicitos. Luego se quiso hacer el verificador simbolico pero para un lenguaje mas complejo (con sentencias, estructuras de control, etc.), y despues de notar que se escapaba de la idea principal de la tesis y de mi tiempo, se encontró el punto medio que era hacer un verificador para modelos simbolicos en un lenguaje simple.

Capítulo 2

Conceptos preliminares

En el diseño de software y hardware para sistemas complejos, cada vez es más el tiempo y esfuerzo dedicado a la verificación en vez de la construcción. Se buscan técnicas para reducir y facilitar el trabajo de la verificación y a la vez incrementar su cobertura. Los métodos formales ofrecen un gran potencial para obtener una integración temprana de la verificación en el proceso de diseño, para proveer técnicas de verificación mas efectivas, y para reducir el tiempo de verificación en general.

La verificación de modelos o *m*odel *c*hecking es una técnica automática de verificación de propiedades sobre sistemas con una cantidad finita de estados. Es una alternativa interesante con respecto al testing o las simulaciones ya que a diferencia de estas técnicas, el model checking hace una prueba exhaustiva del sistema, es decir, analiza todas las trazas posibles de la ejecución del sistema en cuestión. Sin embargo, esto trae un problema, esto es el problema de la explosión de estados. Esto ocurre en sistemas con muchas interacciones internas, y que pueden hacer crecer exponencialmente el espacio de estados posibles del sistema, ya que la prueba es exhaustiva no se puede ignorar ningún estado posible. En los últimos años se ha logrado un gran progreso en cómo lidiar con este problema mediante formas más compactas de representar al sistema, como por ejemplo, una representación simbólica del modelo del sistema.

El modelo del sistema generalmente es generado automáticamente desde una descripción del modelo en un un lenguaje similar a alguno de programación como C, Java, etc. Hay que notar que la especificación de la propiedad prescribe lo que el sistema debe y no debe hacer, en cambio la descripción del modelo señala como se comporta el sistema. El verificador de modelos examina todos los estados relevantes del sistema para verificar si satisface o no la propiedad deseada.

El proceso del verificación de modelos consta de varias fases diferenciables [3]:

Modelado: Hay que modelar el sistema en cuestión usando el lenguaje de descripción de modelos del verificador, y formalizar la propiedad que se desea verificar usando el lenguaje de especificación de propiedades.

Ejecución: Ejecutar el verificador para corroborar la validez de la propiedad en el modelo del sistema.

Análisis: Si la propiedad fue satisfecha, verificar la próxima propiedad (si la hay), si en cambio, no fue satisfecha, hay que refinar el modelo y/o la propiedad y finalmente, repetir el proceso.

2.1 Ventajas y desventajas de la verificación de modelos

Ventajas de la verificación de modelos [3]:

- Es un enfoque general de verificación que es aplicable a una gran variedad de areas, como sistemas embebidos, ingenieria de software, y diseño de hardware.
- Soporta verificación parcial, por lo tanto permite enfocarse en las propiedades escenciales primero.
- No es vulnerable a la probabilidad de que un error sea expuesto, en contraste con el testing y la simulación que son técnicas que apuntan a escenarios particulares de error.
- Provee información en caso de que una propiedad sea inválida, lo çuál es útil para el *debugging*.
- La utilización de un verificador necesita poca interacción con el usuario.
- Esta creciendo en popularidad debido a su facilidad de uso.
- Tiene una base matemática, se basa en teoria de grafos, estructuras de datos y lógica, lo cuál la hace confiable.

Desventajas de la verificación de modelos [3]:

- Es principalmente apropiada para aplicaciones dirigidas por control y menos apropiada para aplicaciones dirigidas por datos ya que los datos suelen pertenecer a dominios infinitos.
- Su aplicación esta sujeta a problemas de decidibilidad; para sistemas de estados infinitos, la verificación de modelos en general no es efectivamente computable.

- Verifica un modelo del sistema, en vez del sistema real mismo, cualquier resultado obtenido es tan bueno como el modelo del sistema.
- Solo verifica los requisitos que se hayan descripto, es decir, no hay garantia de completitud.
- Sufre del problema de la explosión de estados, ya que la cantidad de estados necesarios para modelar el sistema puede facilmente exceder la memoria disponible de la computadora. Si bien hay formas de combatir este problema (ver capítulo 3), los modelos de sistemas reales aun asi suelen ser demasiado grandes.
- Su utilización requiere conocimiento sobre como abstraer un sistema para obtener un modelo adecuado, ademas de conocimientos sobre la lógica utilizada para la especifición de propiedades.
- No garantiza resultados correctos, como cualquier herramienta, el verificador puede contener errores de software.
- En general, verificar sistemas con una cantidad arbitraria de componentes, o sistemas parametrizados, no es posible.

2.2 Modelado de sistemas

En esta sección veremos cómo representar un modelo explícitamente mediante una estructura de Kripke, en el siguiente capítulo veremos otra forma de representación llamada simbólica que representa el modelo mediante una fórmula lógica de primer orden.

Sea AP un conjunto de proposiciones atómicas, una estructura de Kripke M sobre AP es una cuatro-upla $M = (S, S_0, R, L)$ donde [4]:

1. S es un conjunto finito de estados.
2. $S_0 \in S$ es el conjunto de estados iniciales.
3. $R \in S \times S$ es una relación de transición total, es decir para cada estado $s \in S$ existe un estado $s' \in S$ tal que $R(s, s')$ vale.
4. $L\colon S \to 2^{AP}$ es una función que etiqueta a cada estado con el conjunto de proposiciones atómicas que son verdaderas en ese estado. Un camino en la estructura M desde un estado s es una secuencia infinita de estados $p = s_0, s_1, s_2, s_3, ...$, tal que $s = s_0$ y $R(s_i, s_{i+1})$ vale para todo $i > 0$.

Sea $V = v_1, v_2, ..., v_n$ el conjunto de variables del sistema y sea D el dominio, llamaremos una valuación de V a una función que asocia a cada variable de V un valor de D.

Un estado del sistema se puede representar como una valuación de las variables del sistema. Una proposición atómica de la forma $v = d$ donde $v \in V$ y $d \in D$ será verdadera en un estado s si y solo si $s(v) = d$. Dada una valuación, podemos escribir una fórmula que sea verdadera precisamente para esa valuación, por ejemplo si tenemos $V = \{x, y, z\}$ y la valuación $(x \leftarrow True, y \leftarrow True, z \leftarrow False)$ entonces derivamos la fórmula $(x \wedge y \wedge !z)$. En general, una fórmula puede ser verdadera para varias valuaciones. Si adoptamos la convención de que una fórmula representa el conjunto de todas las valuaciones que la hacen verdadera, entonces podremos describir ciertos conjuntos de estados como fórmulas de primer orden. En particular, el conjunto de los estados iniciales del sistema puede describirse como una fórmula de primer orden S_0 sobre las variables en V. Una transición del sistema se puede representar como un par ordenado de valuaciones, de forma similar podemos describir conjuntos de transiciones mediante una formula para ese par, pero para poder expresar la fórmula se necesita una copia V' de V para hablar de siguiente estado, en V' todas las variables estan primadas. Por ejemplo si tenemos una transición $(x \leftarrow True, y \leftarrow True, z \leftarrow False, (x \leftarrow True, y \leftarrow True, z \leftarrow True))$, podemos derivar la fórmula $(x \wedge y \wedge !z \wedge x' \wedge y' \wedge z')$.

Consideremos el siguiente ejemplo, tenemos $V = \{x, y, z\}$ y $D = \{True, False\}$, $S_0(x, y, z) = (x = True \wedge y = True \wedge z = False)$, y tenemos solo una transición: $z := x \wedge y$,consideremos False = 0 y True = 1 por una cuestión de facilidad de lectura. Definimos asi la estructura de kripke (2.1) de la siguiente manera:

$$S = D \times D \times D$$
$$S_0 = \{(1, 1, 0)\}$$
$$R = \{((1, 1, 0), (1, 1, 1)), ((1, 1, 1), (1, 1, 1))\}$$
$$L(1, 1, 0) = \{x = 1, y = 1, z = 0\},$$
$$L(1, 1, 1) = \{x = 1, y = 1, z = 1\}$$

El único camino posible en esta estructura partiendo del estado inicial es: (1, 1, 0), (1, 1, 1), (1, 1, 1), (1, 1, 1) ...

2.3 Especificación de propiedades

Ahora describiremos una lógica para especificar propiedades deseadas en una estructura de Kripke u otra máquina de transición de estados. La lógica utiliza proposiciones atómicas y operadores como la disyunción y la negación para construir expresiones más complicadas que describan propiedades sobre estados. La

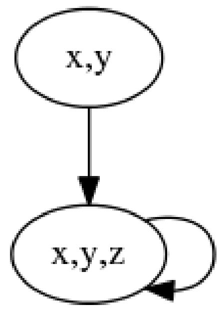

Figure 2.1: Estructura de Kripke para este ejemplo.

lógica temporal es un formalismo que permite describir secuencias de transiciones entre estados en un sistema reactivo, nos interesa saber si en algún momento se llega a un estado determinado o que nunca se llegue a un deadlock. Para esto introduce nuevos operadores especiales que permiten hablar sobre tiempo. Estos operadores pueden combinarse con los operadores lógicos conocidos. Con el propósito de familiarizarnos con las lógicas temporales, introduciremos ahora las lógicas LTL (Linear Temporal Logic), CTL (Computational Tree Logic), y CTL*.

2.3.1 CTL

CTL tiene dos clases de fórmulas, fórmulas de estado y fórmulas de camino. Las primeras son aserciones respecto de las proposiciones atómicas en los estados y de su estructura de ramificación, mientras que las formulas de camino expresan propiedades temporales de caminos. Las fórmulas de estado CTL sobre el conjunto AP de proposiciones atómicas, están formadas de acuerdo a la siguiente gramática[3]:

$$\Phi ::= true|a|\Phi_1 \wedge \Phi_2|\neg\Phi|\exists\varphi|\forall\varphi$$

donde $a \in AP$ y φ es una fórmula de camino. La sintaxis de fórmula de camino CTL esta dada por la siguiente gramática:

$$\varphi ::= \bigcirc\Phi|\Phi_1 \cup \Phi_2$$

donde φ es una fórmula de camino, y Φ, Φ_1, Φ_2 son fórmulas de estado.

Intuitivamente $\exists f$ vale en un estado s, ssi a partir del mismo existe un camino donde vale f (notar que f solo puede ser una formula de estado). Naturalmente $\forall f$ significa que en todos los caminos vale f. En cuanto a $\bigcirc f$, esto vale en el camino

p ssi en el próximo estado (en un camino, existe un único próximo estado) vale f, y $f \cup g$ vale en p solo ssi f vale siempre hasta que valga g. Los operadores de camino solo toman fórmulas de estado.

Los operadores booleanos $false$, $\vee$ y $\rightarrow$ se definen de la manera usual. Las modalidades temporales "eventualmente" y "siempre" pueden ser derivadas de la siguiente manera:

eventualmente:

$$\begin{aligned} \exists\Diamond\Phi &= \exists(true \cup \Phi) \\ \forall\Diamond\Phi &= \forall(true \cup \Phi) \end{aligned}$$

siempre:

$$\begin{aligned} \exists\Box\Phi &= \neg\forall\Diamond\neg\Phi \\ \forall\Box\Phi &= \neg\exists\Diamond\neg\Phi \end{aligned}$$

Ejemplos de fórmulas CTL:

$\forall\Box(\neg crit_1 \vee \neg crit_2)$ (exclusión mutua).
$\forall\Box(amarillo \vee \forall \bigcirc \neg rojo)$ (cada fase de luz roja es precedida por una fase de luz amarilla").

2.3.2 CTL*

CTL* es una lógica que permite combinar libremente los operadores de CTL, por lo cuál CTL* incluye a CTL. Las fórmulas de estado CTL* sobre el conjunto AP de proposiciones atómicas, están formadas de acuerdo a la siguiente gramática[3]:

$$\Phi ::= true|a|\Phi_1 \wedge \Phi_2|\neg\Phi|\exists\varphi$$

donde $a \in AP$ y φ es una fórmula de camino. La sintaxis de fórmula de camino CTL* esta dada por la siguiente gramática:

$$\varphi ::= \Phi|\varphi_1 \wedge \varphi_2|\neg\varphi| \bigcirc \varphi|\varphi_1 \cup \varphi_2$$

donde Φ es una fórmula de estado, y φ, φ_1, φ_2 son fórmulas de camino.

Analizaremos a continuación una lógica temporal llamada cálculo-μ, este puede codificar (y de hecho incluye) a CTL y CTL* [5].

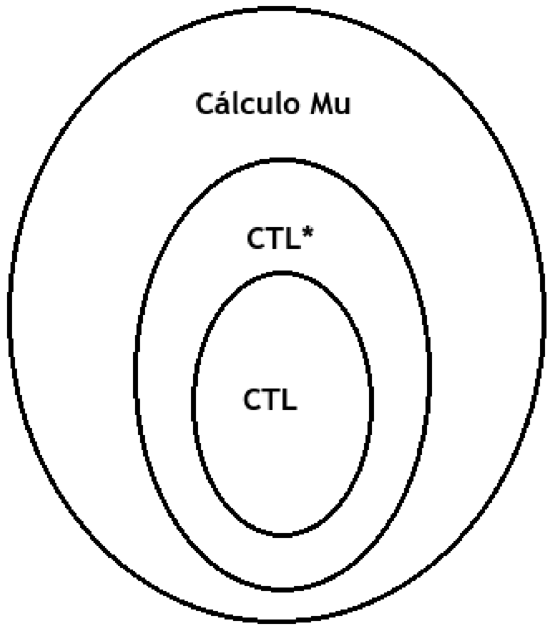

Figure 2.2: Inclusión de lógicas temporales.

2.4 Cálculo-μ

El Cálculo-μ es un poderoso lenguaje para expresar propiedades de sistemas de transición de estados al usar operadores de punto fijo. El Cálculo-μ ha generado mucho interés entre investigadores en verificación asistida por computadoras. Este interés surge del hecho de que muchas lógicas temporales pueden ser codificadas por el Cálculo-μ. Otra fuente de interés en el Cálculo-μ viene de la existencia de algoritmos eficientes de verificación de modelos para este formalismo. Como consecuencia, los procedimientos de verificación para muchas lógicas temporales y modales pueden ser descriptas al traducirse al Cálculo-μ. Hay varias versiones del Cálculo-μ, concretamente usaremos la versión proposicional de Kozen[6].

2.4.1 Sintaxis

Sea $M = (S, T, L)$ una estructura de Kripke y sea $VAR = Q, Q1, Q2, \ldots$ un conjunto de variables relacionales, donde a cada variable relacional se le puede asignar un subconjunto de S, construimos una μ-fórmula como sigue:

Si $p \in AP$, entonces p es una fórmula.
Si $Q \in VAR$, entonces Q es una fórmula.
Si f y g son fórmulas, entonces $\neg f$, $f \vee g$, y $f \wedge g$ son fórmulas.
Si f es una fórmula, entonces $\Box f$ y $\Diamond f$ son fórmulas.
Si $Q \in VAR$ y f es una fórmula entonces $\mu Q.f$ y $\nu Q.f$ son fórmulas.

Las variables pueden estar libres o ligadas en una fórmula a través de un operador de punto fijo. Una fórmula cerrada es una fórmula sin variables libres.

2.4.2 Semántica

El significado intuitivo de $\Diamond f$ es "Es posible realizar una transición a un estado donde f vale", similarmente $\Box f$ significa "f vale en todos los estados alcanzables por medio de una transición" Los operadores μ y ν expresan puntos fijos menores y mayores respectivamente. El conjunto vacío de estados se denota con $False$ y el conjunto de todos los estados S se denota con $True$.

Ejemplos:

$\nu Z \cdot f \wedge \Box Z$ se interpreta como "f es verdadera siempre en todo camino".
$\mu Z \cdot f \vee \Diamond Z$ se interpreta como "existe un camino hacia un estado donde f vale".
$\nu Z \cdot \Diamond True \wedge \Box Z$ se interpreta como "no hay estados que no tengan transiciones hacia otros estados".

Formalmente, una fórmula f se interpreta como un conjunto de estados donde f es verdadera, escribimos este conjunto como $[[f]]$ sobre un sistema de transición de estados M y un ambiente $e : VAR \rightarrow 2^S$, denotaremos $e[Q \leftarrow W]$ como un ambiente que es igual a e solo que Q ahora tiene el valor W. el conjunto $[[f]]$ sobre M y e se define recursivamente de la siguiente manera:

$$
\begin{aligned}
[[p]]\ M\ e &= \{s \mid p \in L(s)\} \\
[[Q]]\ M\ e &= e(Q) \\
[[\neg f]]\ M\ e &= S \setminus [[f]]\ M\ e \\
[[f \wedge g]]\ M\ e &= [[f]]\ M\ e \cap [[g]]\ M\ e \\
[[f \vee g]]\ M\ e &= [[f]]\ M\ e \cup [[g]]\ M\ e \\
[[\Diamond f]]\ M\ e &= \{s \mid \exists t : s \to t \wedge t \in [[f]]\ M\ e\} \\
[[\Box f]]\ M\ e &= \{s \mid \forall t : s \to t \to t \in [[f]]\ M\ e\}
\end{aligned}
$$

$[[\mu Q.f]]\ M\ e$ es el menor punto fijo del predicado transformador $t : 2^S \to 2^S$ definido como $t(W) = [[f]]\ M\ e[Q \leftarrow W]$
$[[\nu Q.f]]\ M\ e$ es el mayor punto fijo del predicado transformador $t : 2^S \to 2^S$ definido como $t(W) = [[f]]\ M\ e[Q \leftarrow W]$

Observemos algunos ejemplos, supongamos que tenemos una estructura de Kripke $M = (S, T, L)$ como la de la figura 2.3, donde $L(s_0) = \{p, q, r\}$, $L(s_1) = \{p, q\}$, $L(s_2) = \{q, r\}$ y $L(s_3) = \{r\}$. Algunos ejemplos de propiedades que uno podria querer verificar son $q \wedge r$, $\Diamond q$, $\Box r$, $\mu Q.(r \wedge \neg p \vee \Diamond Q)$, $\nu Q.(p \wedge \Box Q)$. $q \wedge r$ se cumple en $\{s_2\}$, $\Diamond q$ se cumple en $\{s_0, s_1, s_3\}$, $\Box r$ se cumple en $\{s_1, s_2, s_3\}$, $\mu Q.(r \wedge \neg p \vee \Diamond Q)$ se cumple en S, y $\nu Q.(p \wedge \Box Q)$ se cumple en $\oslash$. Para entender cómo funcionan los operadores de punto fijo consideremos la fórmula $\nu Q.(p \wedge \Box Q)$, es decir, queremos saber que estados cumplen con la propiedad de que p vale siempre en todo camino. Cada iteración del operador esta ilustrada en la figura 2.4. Q se inicializa en True, luego en cada iteración se hace una aproximación al resultado verdadero, los resultados de cada iteración en este caso son, en orden, S, $\{s_0, s_1\}$, $\{s_0\}$, $\oslash$. Como se puede apreciar en el resultado, no hay estado donde valga esta propiedad. Si, en vez de usar el mayor punto fijo usaramos el menor, el procedimiento es análogo, solo que la variable se inicializaria en False, aunque no es dificil darse cuenta que no tiene mucho sentido verificar $\mu Q.(p \wedge \Box Q)$.

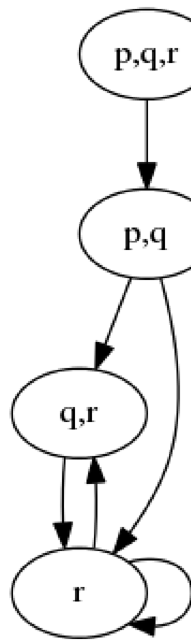

Figure 2.3: Ejemplo de Estructura de Kripke.

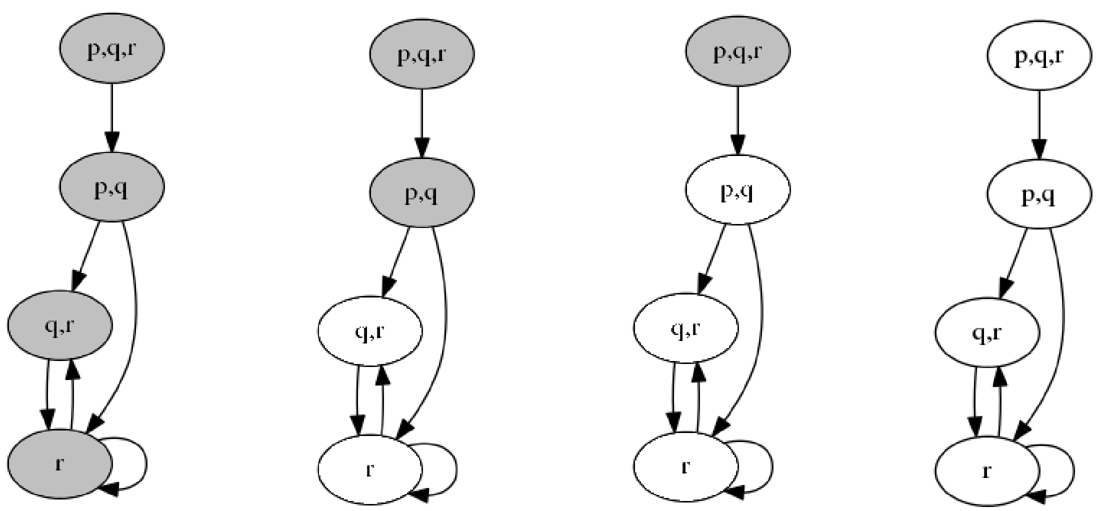

Figure 2.4: Ejecución de operador de punto fijo mayor.

2.4.3 Algoritmo de verificación de modelos explícitos

El algoritmo de verificación de modelos explícitos para Cálculo-μ que analizaremos es el más intuitivo (hay algoritmos mas eficientes), se basa en la semántica anterior, y calcula el subconjunto de estados de un modelo M que cumplen con una formula f. El algoritmo trabaja en forma bottom-up a través de la fórmula, evaluandola a partir de los valores de sus subfórmulas. Este algoritmo requiere tiempo $O(n^k)$, donde n es la cantidad de estados del sistema y k es el número de operadores de punto fijo anidados en la fórmula. Esto se debe a que cada operador de punto fijo hace $n + 1$ iteraciones cómo máximo, ya que en cada iteración se hace una aproximación del resultado ya sea agregando o quitando al menos un estado(menor y mayor punto fijo respectivamente). En la figura 2.5 se puede ver el algoritmo [4], donde f es la fórmula y e es el ambiente de variables relacionales.

```
function eval(f, e)

if f = p then return {s | p ∈ L(s)};
if f = Q then return e(Q);
if f = g1 ∧ g2 then
        return eval(g1, e) ∩ eval(g2, e);
if f = g1 ∨ g2 then
        return eval(g1, e) ∪ eval(g2, e);
if f = ⟨a⟩g then
        return { s | ∃t [s -a-> t and t ∈ eval(g, e)] };
if f = [a]g then
        return { s | ∀t [s -a-> t implies t ∈ eval(g, e)] };

if f = μQ.g(Q) then
        Q_val := False;
        repeat
                Q_old := Q_val;
                Q_val := eval(g, e [Q ← Q_val]);
        until Q_val = Q_old,
        return Q_val;
end if;

if f = νQ.g(Q) then
        Q_val := True;
        repeat
                Q_old := Q_val;
                Q_val := eval(g, e [Q ← Q_val]);
        until Q_val = Q_old,
        return Q_val;
end if;
end function
```

Figure 2.5: Algoritmo de verificación de modelos explícitos para Cálculo-μ.

Capítulo 3

Verificación simbólica de modelos

El algoritmo de verificación de modelos con estados explícitos para Cálculo-μ presentado anteriormente tiene un problema, es muy susceptible a que ocurra una explosión en el tamaño del modelo, especialmente si el grafo de transición de estados se extrae de un sistema concurrente con muchos componentes. En muchos casos, la "complejidad" del espacio de estados es mucho menor que lo que el número de estados indica. A menudo, los sistemas con un gran número de componentes tienen una estructura regular que sugeriria una regularidad correspondiente en el grafo de estados. En consecuencia, existen representaciones mas sofisticadas que explotan esta regularidad. Un buen candidato para tal representación simbólica es el diagrama de decisión binario (BDD) [7]. En esta sección se describe un algoritmo de verificación de modelos simbólicos para Cálculo-μ que opera sobre estructuras de Kripke, esta vez representadas no de manera explícita, sino de manera simbólica a través de fórmulas lógicas (que internamente operan como BDDs).

3.1 Representación de fórmulas lógicas

Los árboles binarios de decisión ordenados (OBDDs) son formas canónicas de representación de fórmulas lógicas. Son considerablemente mas compactos que las formas normales tradicionales como la forma normal conjuntiva y la forma normal disyuntiva, y pueden ser manipulados eficientemente. Por esto, los OBDDs han sido utilizados ampliamente para una variedad de aplicaciones en el diseño asistido por computadoras, incluyendo simulacion simbólica, verificación de lógica combinatoria y, mas recientemente, verificación de sistemas concurrentes con estados finitos.

Para entender la necesidad de usar OBDDs, consideremos primero los árboles binarios de decisión. Un arbol binario de decisión es un árbol dirigido con raiz que consiste en vertices terminales y no terminales. Cada vertice no terminal v esta etiquetado por una variable $var(v)$ y tiene dos hijos: $izq(v)$ corresponde al caso en que v tenga el valor 0 y $der(v)$ en caso contrario. Cada vértice terminal esta etiquetado por una constante $valor(v)$ la cual es 0 o 1. Un árbol binario de decisión para la fórmula $f(a, b, c) = (a \wedge b) \vee (a \wedge c)$ es mostrado en la figura 3.1. Uno puede decidir si una asignación particular a las variables hace verdadera la fórmula o no al atravesar el árbol desde la raíz hasta un vertice terminal. Por ejemplo, la asignación $\{a \leftarrow 1, b \leftarrow 0, c \leftarrow 0\}$ lleva al vértice terminal 0, por lo tanto la fórmula es falsa para esta asignación.

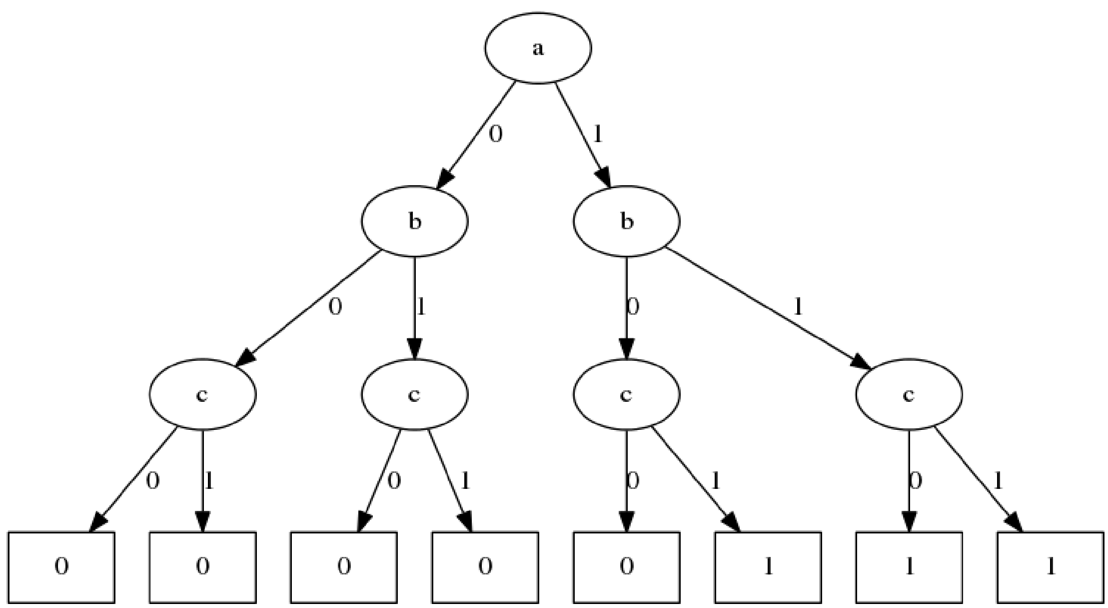

Figure 3.1: Árbol binario de decisión para este ejemplo.

Los árboles binarios de decisión no proveen una representación muy concisa para las funciones lógicas. De hecho, tienen el mismo tamaño que las tablas de verdad. Afortunadamente, es común que haya mucha redundancia en tales árboles. Por ejemplo, en la figura 3.1 todos los caminos donde a tiene el valor 0 llevan al nodo terminal 0, por lo tanto no seria necesario analizar los valores de b y c en esta rama. Esto lleva a pensar que hay formas de reducir el tamaño del árbol unificando subárboles isomorfos. Esto da como resultado un grafo acíclico dirigido (DAG) llamado diagrama binario de decisión (BDD). Mas precisamente, un BDD es un grafo con raiz, dirigido y acíclico con dos tipos de vertices, vertices terminales y no terminales. Estos tienen el mismo significado que en el caso de los árboles. Cada BDD B con raiz v determina una función lógica $f_v(x_1, ..., x_n)$ de la siguiente manera [4]:

1. Si v es un vértice terminal: (a) Si $valor(v) = 1$ entonces $f_v(x_1, ..., x_n) = 1$. (b) Si $valor(v) = 0$ entonces $f_v(x_1, ..., x_n) = 0$.
2. Si v es un vértice no terminal con $var(v) = x_i$ entonces f_v es la función

$$f_v(x_1, ..., x_n) = (\neg x_i \wedge f_{izq(v)}(x_1, ..., x_n)) \vee (x_i \wedge f_{der(v)}(x_1, ..., x_n))$$

Bryant [8] demostró como obtener una representación canónica para funciones lógicas al poner dos restricciones sobre los BDDs. Primero, las variables deberian aparecer en el mismo orden a lo largo de cada camino desde la raiz a un terminal. Segundo, no deberia haber subárboles isomórficos o vertices redundantes en el diagrama. El primer requisito se logra al imponer un ordenamiento total $<$ sobre las variables que etiquetan los vertices en el BDD y requiriendo eso para cada vertice u en el diagrama, si u tiene un sucesor no terminal, entonces $var(u) < var(v)$. El segundo requisito se logra al aplicar repetidamente tres reglas de transformación que no alteran la función representada por el diagrama,

- Eliminar terminales duplicados: Dejar solo un terminal para cada valor y redirigir todos los arcos a los eliminados hacia este.

- Eliminar no terminales duplicados: Si dos no terminales u y v tienen $var(u) = var(v), izq(u) = izq(v) y der(u) = der(v)$, entonces eliminar u o v y redirigir todos los arcos que iban al vertice eliminado hacia el otro.

- Eliminar tests redundantes: Si el no terminal v tiene $izq(v) = der(v)$, entonces eliminar v y redirigir todos los arcos a $izq(v)$.

Empezando por un BDD satisfaciendo la propiedad de ordenamiento, la forma canónica se obtiene aplicando las reglas de transformación hasta que el tamaño del diagrama no pueda ser reducido. Al resultado lo vamos a llamar OBDD.

Hay que destacar que el tamaño del OBDD depende fuertemente del ordenamiento de las variables. Esto se puede ver en las figuras 3.2 y 3.3

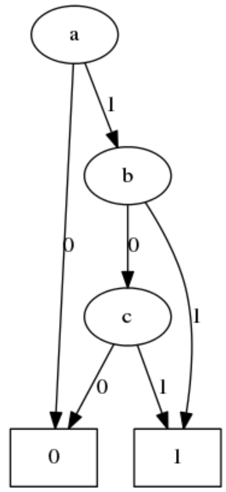

Figure 3.2: OBDD con ordenamiento a < b < c para el ejemplo dado.

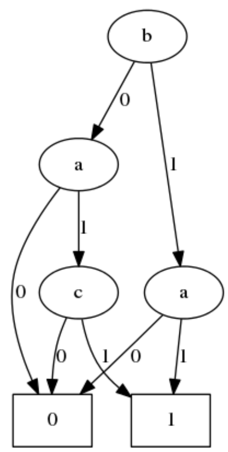

Figure 3.3: OBDD con ordenamiento b < a < c para el ejemplo dado.

3.2 Representación de estructuras de Kripke

Para ilustrar como los OBDDs pueden ser usados para representar concisamente una estructura de Kripke, consideremos la estructura de dos estados mostrada en la figura 3.4 donde $L(s_1) = \{x, y\}$ y $L(s_2) = \{x, y, z\}$. En este caso hay tres variables de estado, x, y, y z. Introducimos tres variables más, x', y' y z', para representar el estado sucesor. Asi, representaremos la transición de s_1 a s_2 como la conjunción

$(x \wedge y \wedge x' \wedge y' \wedge z')$

La fórmula lógica para todo el sistema de transiciones esta dada por

$(x \wedge y \wedge \neg z \wedge x' \wedge y' \wedge z') \vee (x \wedge y \wedge \neg z \wedge x' \wedge y' \wedge \neg z') \vee (x \wedge y \wedge z \wedge x' \wedge y' \wedge \neg z') \vee (x \wedge y \wedge z \wedge x' \wedge y' \wedge z')$

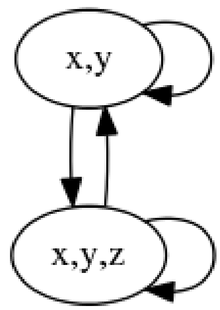

Figure 3.4: Estructura de Kripke con dos estados.

En la fórmula hay cuatro disyuntos porque la estructura de Kripke tiene cuatro transiciones. Esta fórmula se convierte ahora en OBDD para obtener una representación concisa de la relación de transición. En muchos casos, construir una representación explicita de la estructura de Kripke M y luego codificarla como se vió anteriormente no es factible porque la estructura es demasiado grande, incluso si la representación simbólica termina siendo concisa. Por lo tanto, en la práctica construimos los OBDDs directamente desde una descripción de alto nivel del sistema. En el próximo capitulo, cuando introduzcamos el lenguaje de modelado de sistemas utilizado en este trabajo, veremos como se realiza esta transformación.

3.3 Algoritmo de verificación de modelos simbólicos

En la sección anterior vimos como codificar una estructura de Kripke como un OBDD, ahora mostraremos una adaptación del algoritmo de verificación con estados explicitos, ahora con un modelo simbolico en forma de OBDD. En este caso transformaremos las fórmulas del Cálculo-μ de la siguiente manera[4]:

- Los valores Falso y Verdadero estan representados por OBDD-FALSE y OBDD-TRUE que son constantes(hojas).
- Cada proposición atómica p tiene un OBDD asociado con el mismo. Lo denotaremos como $OBDD_p(x)$, donde x es un estado del sistema, este OBDD cumple la propiedad de que x satisface $OBDD_p$ si y solo si $x \in L(p)$.
- El sistema de transiciones tiene un diagrama ordenado de decisión $OBDD_m(x, x')$ asociado al mismo. Un par $(x, x') \in S \times S$ satisface $OBDD_m$ si y solo si $(x, x') \in T$

Asumamos que tenemos una fórmula f de cáluclo-μ con las variables relacionales libres $Q_1, ..., Q_k$. La función $assoc[Q_i]$ asigna a cada variable relacional el OBDD correspondiente al conjunto de estados asociados a esa variable. La notación $assoc(Q \leftarrow B_Q)$ significa que a Q se le da el valor B_Q, se puede ver a $assoc$ como un ambiente de OBDDs. A continuación se da el procedimiento B que, apartir de una fórmula f y una función $assoc$, retorna el OBDD correspondiente a la semántica de f.

$$
\begin{aligned}
B(p, assoc) &= OBDD_p(x)\\
B(Q_i, assoc) &= assoc[Q_i]\\
B(\neg f, assoc) &= \neg B(f, assoc)\\
B(f \wedge g, assoc) &= B(f, assoc) \wedge B(g, assoc)\\
B(f \vee g, assoc) &= B(f, assoc) \vee B(g, assoc)\\
B(\Diamond f, assoc) &= \exists x' : OBDD_m(x, x') \wedge B(f, assoc)(x')\\
B(\Box f, assoc) &= B(\neg\Diamond\neg f, assoc)\\
B(\mu Q.f, assoc) &= FIX(f, assoc, OBDD - FALSE)\\
B(\nu Q.f, assoc) &= FIX(f, assoc, OBDD - TRUE)
\end{aligned}
$$

Donde $B(f, assoc)(x')$ reemplaza cada aparición de x por x', y FIX es la siguiente función [4]:

```
function FIX(f, assoc, B_Q)

result-bdd = B_Q,
repeat
        old-bdd := result-bdd;
        result-bdd := B(f, assoc(Q ← old-bdd));
until (equal(old-bdd, result-bdd));
return(result-bdd);
end function
```

Figure 3.5: Pseudocódigo e la función FIX.

Las operaciones lógicas que ocurren del lado derecho de las ecuaciones son operaciones de BDDs. El significado de la cuantificación existencial de una variable l'ogica esta dada por la siguiente ecuación [9] :

$$\exists x.t = t[0/x] \vee t[1/x]$$

Donde $t[v/x]$ es t pero donde x toma el valor v.

3.3.1 Complejidad

Un interrogante importante en cuanto a la verificación de modelos para cálculo-μ es su complejidad. Los algoritmos mas eficientes conocidos son exponenciales en cuanto al tamaño de la fórmula. Existe la conjetura[4] deque no hay un algoritmo polinomial para el problema de la verificación de modelos para cálculo-μ. Es posible demostrar que el problema esta en $NP \cap co - NP$. Si el problema fuera NP-completo, entonces NP seria igual a co-NP, lo cuál se cree que no es cierto.

Capítulo 4

Lenguaje MC2

MC2 es el verificador de modelos desarrollado en esta tesis, el mismo toma modelos escritos en un lenguaje que tambien llamaremos MC2, el modelo incluye la descripción del sistema y las propiedades que debe satisfacer en Cálculo-μ. El diseño del lenguaje de modelado se centra en la noción de estructuras de Kripke, es decir que con este lenguaje se puede describir el comportamiento del sistema en términos de transiciones de entre estados, y además especificar las propiedades que se desean verificar sobre el modelo. En este capítulo analizaremos la sintaxis y semántica del lenguaje de modelado y el lenguaje de especificación de propiedades, además observaremos algunos detalles relevantes sobre la implementación concreta.

4.1 Sintaxis

Sean $p \in AP$,$X \in VName$, entonces la sintaxis de MC2 se define con la siguiente gramática:

$$
\begin{aligned}
D :=&\ p\\
&\mid D; D\\
\\
C :=&\ E- > E\\
&\mid C; C\\
\\
E :=&\ p\\
&\mid !p\\
&\mid E, E\\
\\
P :=&\ F\\
&\mid P, P\\
\\
F :=&\ p\\
&\mid\ : X\\
&\mid !F\\
&\mid (F\&F)\\
&\mid (F|F)\\
&\mid\ <> F\\
&\mid []F\\
&\mid \%X.F\\
&\mid \$X.F
\end{aligned}
$$

$$M :=\ vars\ D\ rules\ C\ init\ E\ check\ P$$

Usamos la coma $','$ para separar elementos de una lista de expresiones, y , punto y coma $';'$ para separar elementos de una lista de comandos o de declaraciones. La diferencia es sutil pero es importante destacarla para evitar confusión. Un detalle de implementación muy importante que hace falta destacar es que el parsing de E retorna un ambiente, el cual es una lista de pares (p, v), donde $p \in AP$ y $v \in Bool$. Se puede decir que hay una semántica intermedia para E:

$$
\begin{aligned}
[[p]] &= (p, True)\\
[[!p]] &= (p, False)\\
[[E0, E1]] &= [[E0]] \;++\; [[E1]]
\end{aligned}
$$

De ahora en mas cuando hablemos de E, hacemos referencia a la lista generada anteriormente.

4.2 Semántica

Ya definimos la sintaxis del lenguaje, ahora vamos a definir el significado asociado a esta sintaxis. Primero daremos una semántica informal para que sea mas fácil entender como se usa y como funciona el lenguaje MC2, y después entraremos en detalle con la semántica formal.

4.2.1 Semántica informal

La figura 4.2 muestra un ejemplo de una descripción MC2. Aqui representamos una estructura de Kripke con dos estados s_0,s_1 donde $L(s_0) = \{a, b\}$, $L(s_1) = \{a\}$, y $T = \{(s_0, s_1), (s_1, s_0), (s_1, s_1)\}$ como el de la figura 4.1. En la sección *vars* se declara el conjunto de proposiciones atómicas del modelo. La sección *rules* describe las transiciones del sistema. La sección *init* es donde se señala el valor inicial de las proposiciones atómicas. En la sección *check* se especifican las propiedades que se desean verificar sobre el modelo en el estado *init*.

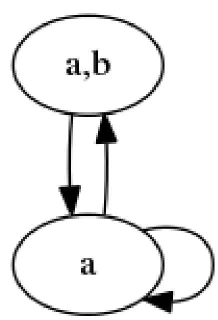

Figure 4.1: Estructura de Kripke del modelo.

```
vars
  a;b

rules
  a,b -> a,!b;
  a,!b -> a,b;
  a,!b -> a,!b

init
  a,b

check
  <>(!a & !b),
  %z.(b | <>:z),
  $z.(b & []:z),
  $z.(a & []:z)
```

Figure 4.2: Ejemplo de descripción MC2.

Se puede ver que las reglas describen precisamente las tres transiciones del sistema. Aquellas proposiciones que no varian su valor de un estado al siguiente, se las puede obviar en la parte derecha de la regla como se ve en la figura 4.3. Esta descripción es equivalente a la anterior. Intuitivamente podemos pensar la parte izquierda de la regla como el estado corriente y la parte derecha como el siguiente estado, pero en realidad podemos representar mas de una transición con una sola regla. Por ejemplo, la descripción de la figura 4.5 modela el sistema de la figura 4.4. Al omitir a en la parte izquierda de las reglas, estamos diciendo que las mismas se cumplen tanto si vale como si no vale a.

```
vars
  a;b

rules
  a,b -> !b;
  a,!b -> b;
  a,!b ->

init
  a,b

check
  <>(!a & !b),
  %z.(b | <>:z),
  $z.(b & []:z),
  $z.(a & []:z)
```

Figure 4.3: Ejemplo de descripción MC2 usando azucar sintáctico.

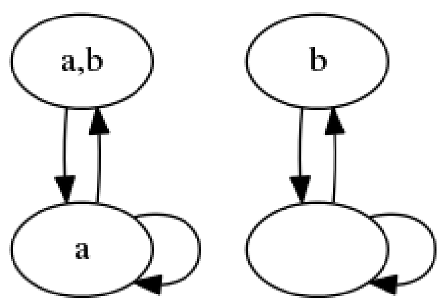

Figure 4.4: Estructura de Kripke del nuevo modelo.

```
vars
  a;b

rules
  b -> !b;
  !b -> b;
  !b ->

init
  a,b

check
  <>(!a & !b),
  %z.(b | <>:z),
  $z.(b & []:z),
  $z.(a & []:z)
```

Figure 4.5: Ejemplo de descripción MC2 con más de una transición por regla.

En la sección *check* se puede ver que hay cuatro propiedades descritas en cálculo-μ, que son las siguientes: $\Diamond(\neg a \wedge \neg b)$, $\mu z.(b \vee \Diamond z)$, $\nu z.(b \wedge \Box z)$, y $\nu z.(a \wedge \Box z)$. Al ejecutar el verificador con la descripción de la figura 4.2, el resultado va a ser el conjunto de propiedades que el modelo haya satisfecho, en este caso $\mu z.(b \vee \Diamond : z)$ y $\nu z.(a \wedge \Box : z)$, ya que existe un camino en donde b vale en algun momento, y a vale siempre en todo camino. Evidentemente $'\$'$ representa a ν y $'\%'$ representa a ν. Cabe destacar también que al anteponer $' :'$ a una cadena estamos haciendo referencia a una variable y no a una proposición.

4.2.2 Semántica formal

En esta sección vamos a formalizar las nociones descriptas en la sección anterior. Vamos a necesitar usar operaciones de OBDDs, para lo cual tenemos NOT, AND, OR, NULL(True si no hay modelos para este OBDD [10]), EXISTS, OBDD-TRUE y OBDD-FALSE. La semántica de una descripción MC2 es la siguiente:

$[[\, vars\, D\, rules\, C\, init\, E\, check\, P]]_m \;=\; [F \mid F \in P \wedge NULL\,(NOT\, inst\, E\,([[F]]_f\, [[C]]_c\, assoc- init))]$

Es decir, de todas las fórmulas en P solo nos quedamos con aquellas que al instanciarlas con los valores de *init* siempre da como resultado $True$. La semántica de una declaración está dada por la siguiente función:

$$\begin{aligned}
[[p]]_d &= (p, False) \\
[[D0; D1]]_d &= [[D0]] \ ++ \ [[D1]]
\end{aligned}$$

Una declaración da como resultado un ambiente, es decir, una lista de proposiciones con sus valores asociados ($False$ en principio). A continuación tenemos la función que denota la semántica de los modelos, un modelo es la disyunción de una o más reglas, a su vez, una regla es una disyunción de todas las transiciones que genera, donde una transición es una conjunción de los OBDDs generados por la evaluación de los ambientes del estado corriente y el siguiente (en el siguiente estado todas las proposiciones deben estar primadas).

$$\begin{aligned}
[[C; D]]_c &= [[C]]_c \ OR \ [[D]]_c \\
[[E0- > E1]]_c &= [[E0]]_e \ AND \ [[E1]]_{e'}
\end{aligned}$$

La evaluación de un ambiente esta dada por las funciones $[[E]]_e$ y $[[E]]_{e'}$, donde la única diferencia entre estas funciones es que la segunda prima a las proposiciones. La semántica de un ambiente da como resultado la conjunción de las proposiciones del mismo con paridad acorde a sus valores asociados.

$$\begin{aligned}
[[(p, True)]]_e &= OBDD_p \\
[[(p, False)]]_e &= NOT \ OBDD_p \\
[[E0 + +E1]]_e &= [[E0]]_e \ AND \ [[E1]]_e
\end{aligned}$$

$$\begin{aligned}
[[(p, True)]]_{e'} &= OBDD_{p'} \\
[[(p, False)]]_{e'} &= NOT \ OBDD_{p'} \\
[[E0 + +E1]]_{e'} &= [[E0]]_{e'} \ AND \ [[E1]]_{e'}
\end{aligned}$$

Por ultimo, la semántica de las fórmulas, es la vista en el capítulo 3. M es el modelo del sistema (un OBDD). Assoc es una función que asocia cada variable relacional con un OBDD. La operación EXISTS de los OBDD toma un conjunto de variables y las elimina existencialmente de un OBDD.

$$
\begin{aligned}
[[p]]_f \ M \ assoc &= OBDD_p \\
[[: X]]_f \ M \ assoc &= assoc \ X \\
[[!F]]_f \ M \ assoc &= NOT \ ([[F]]_f \ M \ assoc) \\
[[F\&G]]_f \ M \ assoc &= ([[F]]_f \ M \ assoc) \ AND \ ([[G]]_f \ M \ assoc) \\
[[F|G]]_f \ M \ assoc &= ([[F]]_f \ M \ assoc) \ OR \ ([[G]]_f \ M \ assoc) \\
[[<> F]]_f \ M \ assoc &= EXISTS \ x' \ : \ M \ AND \ ([[F]]_f(x') \ M \ assoc) \\
[[[]F]]_f \ M \ assoc &= [[! <>!F]]_f \ M \ assoc \\
[[\%X.F]]_f \ M \ assoc &= FIX \ F \ assoc \, OBDD - FALSE \\
[[\$X.F]]_f \ M \ assoc &= FIX \ F \ assoc \, OBDD - TRUE
\end{aligned}
$$

4.3 Diseño e implementación

En esta sección vamos a aclarar detalles del diseño y la implementación del verificador de modelos MC2. La herramienta esta implementada en el lenguaje funcional Haskell, y se interpreta con ghc. La misma está compuesta por los módulos $Types$, Mu, $MuEval$, $Model$, $ModelEval$, $Main$, y usa dos modulos externos, $OBDD$ [10] (provee la estructura con sus operaciones) y $ParseLib$ [11] (tiene utilidades de parsing).

4.3.1 Tipos en MC2

En $MC2$ tenemos proposiciones atómicas (AP) representadas por cadenas, cada una tiene asociada un valor lógico ($True$ o $False$), para lo cual existe un tipo Env (ambiente) que consta de una lista de pares de proposiciones atómicas y sus valores lógicos asociados. Un valor de tipo Env representa el estado del sistema en un momento dado. Tambien definimos el tipo $VName$ como sinónimo de cadenas, pero este tipo lo usamos para hacer referencia a variables relacionales. También hemos definido en este modulo el tipo $Assoc$ como una función Assoc: $VName- > OBDDAP$. $OBDDAP$ hace referencia al tipo de OBDDs donde las variables de sus nodos estan representadas con cadenas (AP). Assoc es un tipo que se utiliza en la semántica de las fórmulas de cálculo-μ, este representa una función que toma el nombre de una variable y devuelve el valor asociado (representado por un OBDD).

4.3.2 Descripción del modelo en MC2

Hay dos modulos dedicados a la descripción del modelo. Uno es $Model$, el cuál contiene la definición de la sintaxis de las declaraciones y comandos, y sus correspondientes $parsers$. El otro módulo es $ModelEval$, este contiene las funciones *ceval*, *deval* y *eeval* correspondientes a los evaluadores de comandos, declaraciones y ambientes respectivamente, además de algunas funciones auxiliares. La función *deval* toma una declaración y un ambiente con proposiciones a inicializar (se usa en la sección *init* unicamente), y a partir de estos genera el ambiente inicial del sistema. La función *eeval* transforma un ambiente en una OBDD-conjunción como se vió en la semántica de ambientes.

4.3.3 Cálculo-μ en MC2

Similarmente tenemos dos modulos dedicados al Cálculo-μ. Uno es Mu, el cuál contiene la definición de la sintaxis, y adicionalmente también contiene el *parser*, y un *printer*. El otro módulo es $MuEval$, este contiene la función *check* que, dada una fórmula, un modelo (OBDD) y la función *Assoc*, evalua la fórmula y devuelve el OBDD correspondiente a su semántica. Al ser funcional la implementación, toda la información necesaria para computar un resultado debe ser pasada como parámetro, por lo que la función *check* también toma dos parámetros extra, necesarios para reescribir los nombres de las proposiciones atómicas del OBDD por sus respectivas versiones primas (cuando es necesario verificar algo sobre el siguiente estado). Además $MuEval$ contiene algunas funciones auxiliares como fix, la cual se utiliza en el cálculo de puntos fijos.

4.3.4 Módulo principal

El módulo $Main$ contiene el *parser* de descripciones MC2, su evaluador y varias funciones auxiliares para la lectura de archivos. El *parser* utiliza los $parsers$ de las declaraciones, comandos (reglas) y propiedades (fórmulas de cálculo-μ). Para el el estado inicial también se usa el *parser* de declaraciones, solo que en vez de inicializarse todas las proposiciones atómicas en $False$, se inicializan como se halla descripto en la sección *init* de la descripción del modelo.

Capítulo 5

Casos de estudio

Ya hemos presentado el verificador de modelos MC2 y el lenguaje de descripción asociado al mismo. Es hora de observar algunos ejemplos concretos de uso de esta herramienta, y ese es el propósito de este capítulo final. Veremos como modelar tres clásicos juegos solitarios de lógica y estrategia, y analizaremos hasta que punto es viable describir manualmente el modelo con esta versión básica de MC2.

5.1 Cruce del rio

Existen muchas versiones de este problema, nosotros modelaremos una de las versiones mas conocidas [12]. El problema es el siguiente, un granjero va a comprar un zorro, un ganso y una bolsa de frijoles. Mientras vuelve a su casa, se encuentra con que debe cruzar un rio, para lo cuál alquila un bote. El granjero solo puede llevar una de sus compras en el barco. Si se deja al zorro con el ganso, el zorro se lo va a comer, y si se deja al ganso con la bolsa de frijoles, este se va a comer los frijoles. El desafio del granjero es el de llevar todas sus compras intactas al otro lado del rio. El problema consiste entonces en elegir sabiamente que compras trasladar en cada viaje.

La descripción MC2 para el modelo de este problema esta en la figura 5.1. Para modelarlo declaramos cuatro pares de proposiciones atómicas, uno para cada compra y otro para el barco/granjero, las proposiciones marcadas con 1 hacen referencia a que estan del lado *origen* del rio, y las marcadas con 2 hacen referencia a que estan del lado *destino* del rio. En cuanto a las reglas, hay dos grupos de reglas a tener en cuenta para describir el comportamiento del sistema, por un lado se describe lo que pasa cuando se dejan juntos objetos incompatibles (por ejemplo, zorro y ganso), por otro lado tenemos las reglas que definen la acción de viajar de un lado del rio al otro (se puede viajar solo o acompañado por un solo objeto). Hay que destacar que, por ejemplo, si $f1$ y $f2$ son falsas significa que los frijoles se han comido, y ninguna regla puede cambiar estos valores. En este caso, en el estado inicial queremos que valga que tanto el granjero/bote como el zorro, el ganso y los frijoles estan del lado *origen* del rio. Por último, la propiedad que queremos verificar es si existe alguna forma de que, siguiendo las reglas, puedan estar el zorro, el ganso y los frijoles en el lado *destino* del rio. En la figura 5.2 está el OBDD que representa al modelo.

```
vars
  f1;f2;b1;b2;g1;g2;b1;b2

rules
  f1,g1,!b1 -> !g1;
  f2,g2,!b2 -> !g2;
  g1,b1,!b1 -> !b1;
  g2,b2,!b2 -> !b2;

  b1,g1 -> !g1,!b1,g2,b2;
  b2,g2 -> !g2,!b2,g1,b1;
  b1,b1 -> !b1,!b1,b2,b2;
  b2,b2 -> !b2,!b2,b1,b1;
  b1,f1 -> !f1,!b1,f2,b2;
  b2,f2 -> !f2,!b2,f1,b1;
  b1 -> !b1,b2;
  b2 -> !b2,b1

init
  f1,b1,g1,b1

check
  %z.((b2 & (f2 & g2)) | <>:z)
```

Figure 5.1: Descripción de modelo del problema del cruce del rio.

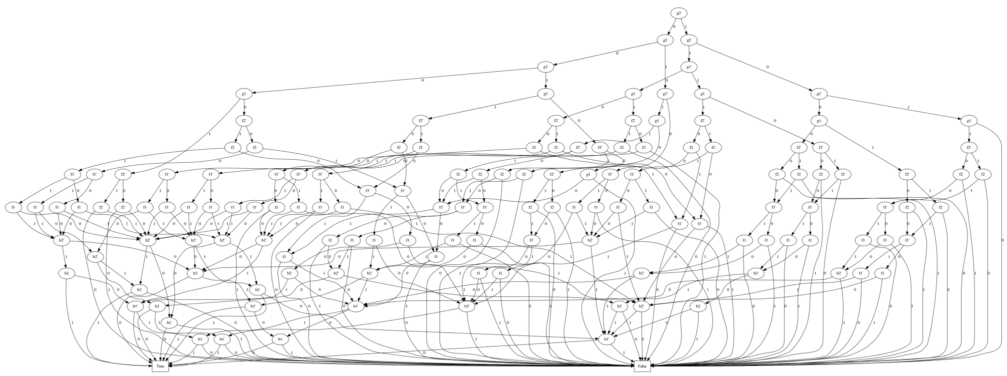

Figure 5.2: OBDD para el modelo del cruce del rio.

5.2 1,2,3, Coloca otra vez

Se parte de una estrella de 5 puntas, ademas de estas, se forman 5 puntos en su interior. Partiendo de uno de los 10 puntos donde no haya una ficha previamente colocada, se cuenta tres posiciones consecutivas sobre una de las aristas que contienen el punto de partida. Tras ello, se coloca una ficha en la tercera posición. En la figura 5.3 podemos ver la estrella, hemos nombrado los puntos externos como $o1, ..., o5$ y los internos como $i1, ..., i5$. Por ejemplo partiendo de $o1$ podemos poner una ficha en $i3$ o $i5$. El conteo puede pasar por una posición en la que haya ficha, pero no puede iniciarse en una posición con ficha. El juego estará resuelto cuando se hayan colocado 9 fichas [13].

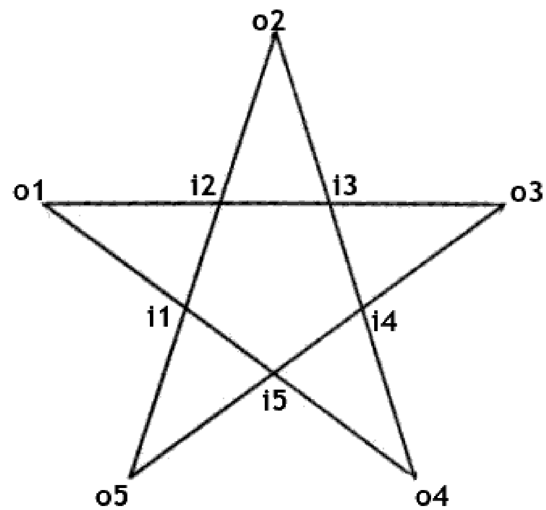

Figure 5.3: Pentagrama de 1,2,3 Coloca otra vez.

En la figura 5.4 se puede ver la descripción MC2 para el modelo de este problema. Para este modelo, declaramos una proposición para cada uno de los diez puntos de la estrella, donde cada proposición es verdadera o falsa dependiendo de si hay o no una ficha en ese punto. Las reglas en este caso representan las jugadas posibles a partir de cada punto. En el estado inicial no hace falta describir nada, ya que inicialmente queremos que no haya ficha en ningun punto de la estrella (todas las proposiciones estan inicializadas en $False$). Por último, la propiedad que queremos verificar es si es posible poner nueve fichas en la estrella, a modo de ejemplo, la formula pide que haya una ficha en todos los puntos excepto $i5$.

```
vars
  o1;o2;o3;o4;o5;i1;i2;i3;i4;i5

rules
  !o1,!i3 -> i3;
  !o1,!i5 -> i5;
  !o2,!i1 -> i1;
  !o2,!i4 -> i4;
  !o3,!i2 -> i2;
  !o3,!i5 -> i5;
  !o4,!i5 -> i5;
  !o4,!i4 -> i4;
  !o5,!i2 -> i2;
  !o5,!i4 -> i4;

  !i1,!o2 -> o2;
  !i1,!o4 -> o4;
  !i2,!o5 -> o5;
  !i2,!o3 -> o3;
  !i3,!o1 -> o1;
  !i3,!o4 -> o4;
  !i4,!o2 -> o2;
  !i4,!o5 -> o5;
  !i5,!o1 -> o1;
  !i5,!o3 -> o3

init

check
  %z.((o1 & (o2 & (o3 & (o4 & (o5 & (i1 & (i2 & (i3 & i4)))))))) | <>:z)
```

Figure 5.4: Descripción de modelo del problema de 1,2,3 Coloca otra vez.

5.3 Ranas saltarinas

En este juego se tiene una tira de papel dividida en siete casillas [13]. La posición inicial es la indicada con tres fichas azules (blancas) y tres rojas (grises) colocadas como en la figura 5.5. El objetivo del juego consiste en permutar las posiciones de las fichas azules y rojas. Es decir, las azules han de pasar a ocupar las posiciones de las rojas y viceversa. Para ello son válidos los siguientes movimientos: - Una ficha puede moverse a un lugar contiguo, si éste está vacío. - Una ficha junto a otra de distinto color puede saltar por encima de ella si el salto (por encima de una sola ficha) le lleva a una casilla vacía. - Son válidos tanto los movimientos hacia atrás como hacia adelante.

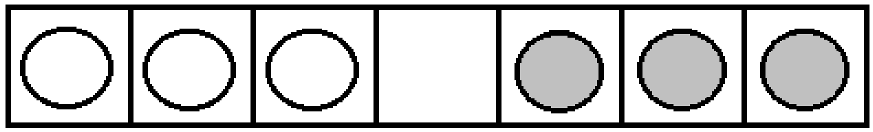

Figure 5.5: Juego de las ranas saltarinas.

La descrición MC2 del modelo para este juego esta en la figura 5.6. En este caso, tenemos siete proposiciones que representan la ocupación de cada casilla por una ficha roja, y siete más para las fichas azules, es de imaginar que una casilla i esta vacia si ri y bi son falsas, donde ri significa que hay una ficha roja en la casilla i, y bi significa que hay una azul en la misma casilla. Las reglas estan divididas en cuatro bloques, dos para cada dirección en la que se puede mover una ficha. Los dos bloques de una dirección representan las primeras dos reglas del juego. Por su puesto, los bloques de la otra dirección son análogos. Como estado inicial tenemos las fichas azules colocadas del lado izquierdo de la tira de casillas, y las rojas del lado derecho, dejando vacia la casilla del medio. Por último, la propiedad que deseamos verificar es si es posible permutar las posiciones de las fichas rojas con las azules y viceversa.

```
vars
  b1;b2;b3;b4;b5;b6;b7;
  r1;r2;r3;r4;r5;r6;r7

rules
  b2,!b1,!r1 -> !b2,b1;
  b3,!b2,!r2 -> !b3,b2;
  b4,!b3,!r3 -> !b4,b3;
  b5,!b4,!r4 -> !b5,b4;
  b6,!b5,!r5 -> !b6,b5;
  b7,!b6,!r6 -> !b7,b6;

  r2,!b1,!r1 -> !r2,r1;
  r3,!b2,!r2 -> !r3,r2;
  r4,!b3,!r3 -> !r4,r3;
  r5,!b4,!r4 -> !r5,r4;
  r6,!b5,!r5 -> !r6,r5;
  r7,!b6,!r6 -> !r7,r6;

  b1,!b2,!r2 -> !b1,b2;
  b2,!b3,!r3 -> !b2,b3;
  b3,!b4,!r4 -> !b3,b4;
  b4,!b5,!r5 -> !b4,b5;
  b5,!b6,!r6 -> !b5,b6;
  b6,!b7,!r7 -> !b6,b7;

  r1,!b2,!r2 -> !r1,r2;
  r2,!b3,!r3 -> !r2,r3;
  r3,!b4,!r4 -> !r3,r4;
  r4,!b5,!r5 -> !r4,r5;
  r5,!b6,!r6 -> !r5,r6;
  r6,!b7,!r7 -> !r6,r7;

  b3,r2,!b1,!r1 -> !b3,b1;
  b4,r3,!b2,!r2 -> !b4,b2;
  b5,r4,!b3,!r3 -> !b5,b3;
  b6,r5,!b4,!r4 -> !b6,b4;
  b7,r6,!b5,!r5 -> !b7,b5;

  r3,b2,!b1,!r1 -> !r3,r1;
  r4,b3,!b2,!r2 -> !r4,r2;
  r5,b4,!b3,!r3 -> !r5,r3;
  r6,b5,!b4,!r4 -> !r6,r4;
  r7,b6,!b5,!r5 -> !r7,r5;

  b1,r2,!b3,!r3 -> !b1,b3;
  b2,r3,!b4,!r4 -> !b2,b4;
  b3,r4,!b5,!r5 -> !b3,b5;
  b4,r5,!b6,!r6 -> !b4,b6;
  b5,r6,!b7,!r7 -> !b5,b7;

  r1,b2,!b3,!r3 -> !r1,r3;
  r2,b3,!b4,!r4 -> !r2,r4;
  r3,b4,!b5,!r5 -> !r3,r5;
  r4,b5,!b6,!r6 -> !r4,r6;
  r5,b6,!b7,!r7 -> !r5,r7


init
  b1,b2,b3,r5,r6,r7

check
  %z.((r1 & (r2 & (r3 & (b5 & (b6 & b7))))) | <>:z)
```

Figure 5.6: Descripción de modelo del problema de las ranas saltarinas.

Nos podemos preguntar como seria la descripción MC2 del modelo si la tira tuviera mas de siete casillas, esta claro que cada vez se vuelve más humanamente inviable a medida que se incrementa el tamaño de casillas. Esto es debido a que la herramienta es de bajo nivel y no cuenta con abstracciones estructurales como los arreglos. Pero es fácil escribir un programa en un lenguaje de alto nivel que genere automáticamente una descripción MC2 para un sistema determinado. En la figura 5.7 se puede ver un pseudocódigo para un *parser* de el problema de las ranas saltarinas para N casillas.

```
int N = 7;
int i = 0;
string res = "vars \n";

for (i = 1; i < N; i++){
  res += "b" + (string)i + ";r" + (string)i + ";"
}
res  += "b" + (string)N + ";r" + (string)N + "\n"
res += "rules \n"
for (i = 1; i < N; i++){
  res += "b" + (string)i + ",!b" + (string)(i+1) + ",!r" + (string)(i+1) + "->" + "!b" + (string)i + ",b" + (string)(i+1)+";";
  res += "r" + (string)i + ",!r" + (string)(i+1) + ",!b" + (string)(i+1) + "->" + "!r" + (string)i + ",r" + (string)(i+1)+";";
}
for (i = 2; i <= N; i++){
  res += "b" + (string)i + ",!b" + (string)(i-1) + ",!r" + (string)(i-1) + "->" + "!b" + (string)i + ",b" + (string)(i-1)+";";
  res += "r" + (string)i + ",!r" + (string)(i-1) + ",!b" + (string)(i-1) + "->" + "!r" + (string)i + ",r" + (string)(i-1)+";";
}
for (i = 1; i < N-1; i++){
  res += "b" + (string)i + ",r" + (string)(i+1) +",!b" + (string)(i+2) + ",!r" + (string)(i+2) + "->" + "!b" + (string)i + ",b" + (string)(i+2)+";";
  res += "r" + (string)i + ",b" + (string)(i+1) +",!r" + (string)(i+2) + ",!b" + (string)(i+2) + "->" + "!r" + (string)i + ",r" + (string)(i+2)+";";
}
for (i = 3; i < N; i++){
  res += "b" + (string)i + ",r" + (string)(i-1) +",!b" + (string)(i-2) + ",!r" + (string)(i-2) + "->" + "!b" +(string) i + ",b" + (string)(i-2)+";";
  res += "r" + (string)i + ",b" + (string)(i-1) +",!r" + (string)(i-2) + ",!b" + (string)(i-2) + "->" + "!r" +(string) i + ",r" + (string)(i-2)+";";
}
res += "b" + (string)N + ",r" + (string)(N-1) +",!b" + (string)(N-2) + ",!r" + (string)(N-2) + "->" + "!b" + (string)N + ",b" + (string)(N-2);
res += "r" + (string)N + ",b" + (string)(N-1) +",!r" + (string)(N-2) + ",!b" + (string)(N-2) + "->" + "!r" + (string)N + ",r" + (string)(N-2);
res += "init \n"
for (i = 1; i < N; i++){
  if (i<N/2){
    res += "b" + (string)i + ",";
  }
  if (i>(N/2+1)){
    res += "r" + (string)i + ",";
  }
}
res += "r" + (string)N;
res += "check \n"
res += "%z.("
for (i = 1; i < N; i++){
  if (i<N/2){
    res += "(r" + (string)i + "&";
  }
  if (i>(N/2+1)){
    res += "(b" + (string)i + "&";
  }
}
res += "b" + (string)N;
for (i = 1; i < N; i++){
  res += ")";
}
res+= "| <>:z)"
```

Figure 5.7: Pseudocódigo de *parser* de alto nivel.

Conclusión

Para el desarrollo de esta tesis, se ha investigado sobre la verificación de modelos, sus variantes como la verificación de modelos explicitos y la verificación de modelos simbólicos, para lo cuál tambien fue necesario investigar sobre lógicas temporales para llegar asi a entender el cálculo-μ y aplicarlo a la práctica.

Se logró desarrollar una nueva herramienta de verificación de modelos para cálculo-μ, con un lenguaje de modelado propio basado reglas de construcción de estructuras de kripke, lo que trae algo nuevo a la mesa de verificadores para cálculo-μ ya existentes como mCRL2 [14], TAPAs [15] y CWB [16].

Como trabajos futuros, se puede considerar la idea de hacer más versatil el lenguaje de modelado, ya sea agregando mas tipos de datos y/o extendiendo las formas de expresar el flujo del sistema que se está modelando.

Bibliografia

[1] E. Clarke, O. Grumberg, and D. Long, *Model Checking*.

[2] P. F. Castro, C. Kilmurray, A. Acosta, and N. Aguirre, *dCTL: A Branching Time Temporal Logic for Fault-Tolerant System Verification*. 2011.

[3] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 2000.

[5] E. A. Emerson, *Temporal and modal logic*. 1995.

[6] D. Kozen, *Results on the prepositional mu-calculus*. Elsevier Science Publishers B.V., 1983.

[7] J. Burch, E. M. Clarke, and K. L. McMillan, *Symbolic Model Checking* 10^{20} *States and Beyond**. LICS, 1990.

[8] R. E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams*. Fujitsu Laboratories, Ltd., 1992.

[9] H. R. Andersen, *An Introduction to Binary Decision Diagrams*. Lecture Notes, IT University of Copenhagen, 1999.

[10] J. Waldmann, *The OBDD package*. https://github.com/jwaldmann/haskell-obdd.

[11] G. Hutton and E. Meijer, *A LIBRARY OF MONADIC PARSER COMBINATORS*. http://www.informatik.uni-bremen.de/cofi/CASL-CD/Tools/Hets/src/Haskell/Hatchet/ParseLib.hs, 2008.

[12] J. Hadley and D. Singmaster, *Problems to Sharpen the Young.* The Mathematical Gazette, 1992.

[13] http://juegosdelogica.net/juegosdeestrategia/.

[14] J. Groote and M. Mousavi, *Modeling and analysis of communicating systems.* The MIT press, 2014.

[15] F. Calzolai, R. D. Nicola, M. Loreti, and F. Tiezzi., *TAPAs: a Tool for the Analysis of Process Algebras.* LNCS, Springer-Verlag, 2008.

[16] F. Moller, *The Edinburgh Concurrency Workbench.* http://homepages.inf.ed.ac.uk/perdita/cwb/.