DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Tesis de Licenciatura

Reducción de subárboles repetidos en *e*CDDs para el *model checking* temporizado

6 de junio de 2009, Buenos Aires, Argentina.

Resumen

Los sistemas de tiempo real son por naturaleza críticos. Sus fallas pueden resultar en serias pérdidas, tanto materiales como también de vidas humanas. Además, en general están descriptos por la interacción de varios componentes y resulta muy difícil asegurar que determinadas propiedades (que representan de alguna manera requisitos o condiciones deseables del sistema) se cumplan.

Hoy en día, existen herramientas denominadas model checkers (por ejemplo, UPPAAL, HyTech, KRONOS) utilizadas para expresar y verificar propiedades sobre este tipo de sistemas. Una de las propiedades más requerida es la de establecer si cierto conjunto de estados del sistema es o no alcanzable desde un estado inicial determinado. Esta verificación es costosa, y a veces prohibitiva, en términos de tiempo y memoria, debido al problema de la explosión combinatoria de estados. Recientemente se ha estudiado el uso de estructuras de decisión como posible alternativa a las representaciones clásicas utilizadas en la verificación, de forma de lograr reducir, en cierto grado, los efectos de tal explosión.

En este trabajo se estudia la presencia de redundancia interna en una estructura de datos presentada en [Pav06], basada en árboles de decisión. Primero cuantificamos dicha redundancia y luego implementamos un método para hacer *aliasing* de las partes repetidas de la estructura, efectivamente reduciendo el espacio consumido.

Alumno:
Gervasio Daniel Perez
Libreta universitaria: 316/01
Correo electrónico: gdperez@dc.uba.ar

Directores:
Dr. Fernando Pablo Schapachnik
Lic. Esteban José Pavese

Agradecimientos

Como en todo proyecto personal tanto académico como de vida, el esfuerzo propio es condición necesaria pero no suficiente para el éxito. Creo que es un espejismo creer que los logros personales son mérito exclusivo de uno. Por eso, es tan importante esta página de agradecimientos como todo el resto de este trabajo; gracias al apoyo y a la colaboración directa o indirecta de muchas personas puedo estar ahora escribiendo estas palabras.

Para empezar, agradezco a mis padres y les dedico esta Tesis, no sólo por el hecho innegable de que son la causa de mi existencia, sino también por su apoyo tanto económico como afectivo en toda las etapas de mi educación, sin olvidar su gran confianza en mis capacidades (a menudo mayor que la mía propia).

No menos importante es agradecer a esta Facultad, extendiéndome a todo el personal docente y no docente. Gracias a ellos tuve una educación gratuita y, a mi entender, de un nivel excepcional. Agradezco a todos los docentes de la carrera, que invariablemente mostraron vocación para la enseñanza y un trato cordial. Las instalaciones y equipos de la Facultad siempre estuvieron disponibles. También agradezco la dedicación de Fer y Esteban, que con su buena onda, conocimientos y recomendaciones fueron esenciales en el buen tránsito de la sinuosa senda que puede ser una Tesis de Licenciatura.

Dejo la anteúltima tanda de agradecimientos para toda la gente que recorrió todo este camino conmigo. A Cristian, Francisco y Federico, por su amistad añeja; a mis amigos de la facu y compañeros de sufrimiento A Reum, Ana, Caro, Emi, Eze, Lu, Majo, Nicos; a mis compañeros del laburo que siempre tuvieron palabras de aliento; a mi familia tanto de sangre como política que siempre se mantuvieron al día en el desarrollo de mi Tesis, especialmente a mi Suegra (la terminé, Lili, la terminé!!!) tanto por su apoyo como por sus tirones de orejas simbólicos (a veces literales) para que me ponga las pilas.

Dejo para lo último a Lala, que con su amor, apoyo, respeto y paciencia, me ayuda a ser cada día un poquito menos peor :)

Índice general

Índice de figuras

Índice de cuadros

List of Algorithms

Capítulo 1

Introducción y definiciones previas

1.1. Motivación

Los sistemas de tiempo real son por naturaleza críticos. Sus fallas pueden poner en peligro la vida humana o representar cuantiosas pérdidas materiales. Además, en general están dados por la interacción de varios componentes y resulta muy difícil asegurar que determinadas propiedades (que representan de alguna manera requisitos o condiciones deseables del sistema) se cumplan.

En las últimas décadas se han desarrollado diversos formalismos tanto para representar los sistemas como para expresar propiedades sobre ellos. En particular, surgieron los autómatas temporizados [Alu92] para describir comportamiento y la lógica TCTL [ACD93] para predicar propiedades sobre éstos. Al poco tiempo aparecieron los primeros *model checkers*, como UPPAAL [BLL+96], HyTech [HHWT95] y KRONOS [BDM+98, DOTY96] para este formalismo, programas que toman la especificación de un sistema y una propiedad sobre él y determinan si ésta se cumple o no. Lamentablemente, el problema de verificar una propiedad sobre un sistema suele ser muy costoso en tiempo y en espacio; a veces prohibitivo.

En los últimos tiempos, se han buscado nuevas formas de representar los estados posibles del sistema, generados durante el proceso de verificación del mismo. Dicha representación resulta crucial en el problema de la explosión combinatoria de estados, ya que se busca abarcar problemas más grandes en menos tiempo. En uno de los tantos esfuerzos para obtener esta reducción, se han planteado interrogantes acerca de la posibilidad de utilizar estructuras de datos diferentes a las que se han utilizado hasta ahora. En particular, se ha comenzado a analizar la posibilidad de utilizar estructuras de decisión para representar los distintos valores que pueden ir tomando los distintos relojes, que modelan el paso del tiempo, en el sistema. Existen algunos resultados teóricos con este tipo de estructuras [Wan03, Wan00, BLP+99], pero aún no hay suficientes resultados prácticos que avalen (o refuten) estas ideas. El desafío que plantean estas estructuras no es trivial: se necesita proveer no sólo la estructura (lo cual presenta problemas de eficiencia tanto en tiempo de ejecución de las operaciones como en espacio utilizado), sino también nuevos algoritmos que resuelvan el problema tal como se lo venía resolviendo, probando además que el método utilizado es correcto.

Para comenzar la presente introducción será necesario primero recorrer una larga lista de definiciones previas que servirán para fijar la nomenclatura. En este capítulo, entonces, presentamos la terminología a utilizar a lo largo del trabajo. Entonces, en las secciones que siguen nos dedicaremos exclusivamente a comentar las definiciones correspondientes al model checking.

Para ello, se tomará como marco la pregunta fundacional de esta disciplina:

> Dado el autómata temporizado G, el conjunto de requerimientos R y la relación $\models$ que expresa *satisfacción*, ¿vale $G \models R$?

En las secciones subsiguientes definiremos formalmente cada uno de los componentes que permitirán responder esta pregunta. Esta introducción tendrá como base aquella presente en [Yov96].

1.2. Autómata temporizado

Para lograr el objetivo de definir un autómata temporizado y su semántica, resulta necesario presentar antes algunos conceptos preliminares estrechamente relacionados.

1.2.1. Relojes

Durante el transcurso de toda esta introducción, $\mathbb{N}$ denotará al conjunto de números naturales, $\mathbb{Z}$ al conjunto de números enteros, $\mathbb{R}$ al conjunto de números reales y $\mathbb{R}^+$ a los reales no negativos. Asimismo, $\mathcal{P}(C)$ denotará al *conjunto de partes del conjunto* C.

Definición 1.2.1 (Relojes, valuaciones sobre relojes) *Sea X un conjunto finito de variables, a las que se denominarán* relojes. *Una* valuación *es una función que asigna un número real no negativo a cada reloj de X. Intuitivamente, una valuación de un reloj es un valor particular de éste.*

El conjunto de valuaciones sobre X se denota como $\mathcal{V}_X$, y está compuesto por las funciones totales que van de X a $\mathbb{R}^+$: $[X \to \mathbb{R}^+]$.

Sea $v \in \mathcal{V}_X$ y $\delta \in \mathbb{R}^+$. Se denota $v + \delta$ la valuación definida de la siguiente forma:

$$(\forall x \in X)(v + \delta)(x) = v(x) + \delta$$

Definición 1.2.2 (Asignaciones sobre relojes) *Sea X^* el conjunto $X \cup \{0\}$. Una* asignación *es una función que asigna a cada reloj el valor de otro reloj, o bien el valor* 0.

El conjunto de asignaciones sobre los relojes de X se denota como Γ_X, y está compuesto por las funciones totales que van de X en X^*: $[X \to X^*]$.

Si $v \in \mathcal{V}_X$ es una valuación y $\gamma \in \Gamma_X$ es una asignación, se denota con $v[\gamma]$ a la valuación definida como:

$$v[\gamma](x) = \begin{cases} v(\gamma(x)) & \text{si } \gamma(x) \in X \\ 0 & \text{si } \gamma(x) = 0 \end{cases}$$

1.2.2. Restricciones sobre relojes

Con el objetivo de poder predicar sobre los valores de los relojes, introducimos a continuación una sintaxis y una semántica para expresar restricciones sobre ellos. Todo ello permitirá comparar los relojes entre ellos y contra valores constantes.

Definición 1.2.3 (Sintaxis de las restricciones sobre relojes) *Dado un conjunto finito de relojes X, se define inductivamente la gramática del lenguaje Ψ_X que permite expresar restricciones sobre los relojes de X de la siguiente manera:*

$$\begin{array}{l} \forall x, x' \in X, c \in \mathbb{N}, d \in \mathbb{Z}: \\ \quad \psi \quad \to \quad c \prec x \quad \mid \quad x \prec c \quad \mid \quad x - x' \prec d \quad \mid \quad \psi \wedge \psi \quad \mid \quad \neg\psi \\ \quad \prec \quad \to \quad < \quad \mid \quad = \end{array}$$

Sin pérdida de generalidad, a partir de este momento sólo haremos referencia a restricciones sobre relojes de longitud finita, que son las que serán de interés en el trabajo.

Definición 1.2.4 (Semántica de las restricciones sobre relojes) *El lenguaje Ψ_X se interpreta sobre las valuaciones. La satisfacibilidad de una restricción $\psi \in \Psi_X$ por una valuación $v \in \mathcal{V}_X$, denotada como $v \models \psi$, se define inductivamente de la siguiente forma:*

$$\begin{array}{l} \forall x, x' \in X, c \in \mathbb{N}, d \in \mathbb{Z}: \\ \begin{array}{lcl} v \models x < c & \Leftrightarrow & v(x) < c \\ v \models x = c & \Leftrightarrow & v(x) = c \\ v \models c < x & \Leftrightarrow & c < v(x) \\ v \models c = x & \Leftrightarrow & c = v(x) \\ v \models x - x' < d & \Leftrightarrow & v(x) - v(x') < d \\ v \models x - x' = d & \Leftrightarrow & v(x) - v(x') = d \\ v \models \psi \wedge \psi' & \Leftrightarrow & v \models \psi' \wedge v \models \psi'' \\ v \models \neg\psi' & \Leftrightarrow & v \not\models \psi' \end{array} \end{array}$$

Definición 1.2.5 (Conjunto característico) *Se llama* conjunto característico *de ψ—y se nota $[\![\psi]\!]$—al conjunto de valuaciones que satisfacen ψ. Formalmente:*

$$[\![\psi]\!] = \{v : v \in \mathcal{V}_X \wedge v \models \psi\}$$

En lo siguiente, será usual referirnos a *zonas temporales* para entender a las restricciones temporales cuyo conjunto característico es un convexo, y *regiones temporales* para hablar de restricciones temporales cuyos conjuntos característicos sean no convexos.

1.2.3. Sintaxis de los autómatas temporizados

Una vez presentados los conceptos anteriores, estamos en condiciones de definir los autómatas:

Definición 1.2.6 (Autómata temporizado) *Un* autómata temporizado *G es una tupla $\langle S, X, E, A, I\rangle$ donde:*

- *$S = \{s_0, \ldots, s_m\}$ es un conjunto finito de* estados discretos *o* locaciones.
- *$X = \{x_0, \ldots, x_n\}$ es un conjunto finito de* relojes.
- *E es un conjunto finito de* etiquetas.
- *A es un conjunto finito de arcos de la forma $\langle s_i, e, cond, asig, s_j\rangle$ donde:*
  - *$s_i \in S$ es el* origen,
  - *$e \in E$ es la* etiqueta,
  - *$cond \in \Psi_X$ es la* condición de activación,
  - *$asig \in \Gamma_X$ es la* asignación *y*
  - *$s_j \in S$ es el* destino.
- *$I \in [S \to \Psi_X]$ (si $s \in S$ decimos que $I(s)$ es el* invariante *del estado discreto s).*

1.2.4. Semántica de los autómatas temporizados

Los autómatas temporizados introducidos anteriormente se interpretan de la siguiente forma:

Definición 1.2.7 (Sistema de transiciones etiquetado) *Dado un autómata temporizado G tal que $G = \langle S, X, E, A, I\rangle$, el significado de G es un* sistema de transiciones etiquetado *infinito $(Q_X, \rightarrow)$ donde Q_X es el* conjunto de estados *y $\rightarrow$ es la* relación de transición *que se define formalmente como sigue:*

- $Q_X = \{(s, v) : s \in S \wedge v \in \mathcal{V}_X \wedge v \models I(s)\}$ *(Es decir, Q_X está compuesto por tuplas de estados discretos y valuaciones de relojes que satisfacen el invariante del estado discreto.)*
- *La relación de transición $\rightarrow \subseteq Q_X \times (E \cup \mathbb{R}^+) \times Q_X$ está definida por las siguientes reglas:*
  - *Transiciones discretas:*

$$\frac{\langle s_i, e, cond, asig, s_j\rangle \in A \quad v \models cond \quad v[asig] \models I(s_j)}{(s_i, v) \rightarrow^e (s_j, v[asig])}$$

    El estado $(s_j, v[asig])$ se denomina sucesor discreto *de (s_i, v) y este último* predecesor discreto *del primero. Intuitivamente, se entiende que a partir de un estado el sistema puede evolucionar a través de un arco que lo conduce a otro estado donde las valuaciones de los relojes son las mismas (i.e., no ha transcurrido tiempo) excepto por aquellos relojes que se ha vuelto a cero o se les ha asignado otro valor (asig). Para que este tipo de transiciones pueda ser tomada las valuaciones del estado original deben satisfacer la condición de activación del arco y las resultantes de la asignación del arco, el invariante del estado destino.*
  - *Transiciones temporales:*

$$\frac{\delta \in \mathbb{R}^+ \quad (\forall \delta')(\delta' \in \mathbb{R}^+ \wedge \delta' \leq \delta \implies v + \delta' \models I(s))}{(s, v) \rightarrow^\delta (s, v + \delta)}$$

    El estado $(s, v + \delta)$ se denomina sucesor temporal *de (s, v) y este último* predecesor temporal *del primero. Intuitivamente, en ellas sólo transcurre tiempo. No hay asignaciones de valores a relojes, ni hay un cambio de estado discreto. El único requisito para poder tomarla es que el invariante del estado discreto se siga cumpliendo durante la duración (transcurso del tiempo) de la transición.*

Definición 1.2.8 (Ejecución) *Sea $G = \langle S, X, E, A, I\rangle$ un autómata temporizado y $(Q, \rightarrow)$ el sistema de transición etiquetado asociado a dicho autómata. Una* ejecución *r de G es una secuencia infinita de aplicaciones de la relación $\rightarrow$ de la forma*

$$\begin{array}{l}(\forall i)(i \in \mathbb{N} \implies q_i \in Q \wedge l_i \in E \cup \mathbb{R}^+) \\ r \quad = \quad q_0 \rightarrow^{l_0} q_1 \rightarrow^{l_1} q_2 \rightarrow^{l_2} \ldots \rightarrow^{l_{i-1}} q_i \rightarrow^{l_i} \ldots\end{array}$$

Luego, si $q \in Q$, se denotará $R_G(q)$ el conjunto de ejecuciones de G tal que $q_0 = q$ y $R_G = \bigcup_{q \in Q} R_G(q)$ al conjunto de ejecuciones de G.

A partir de esta noción de ejecución se define una posición *p de una ejecución r como un par $(i, \delta) \in \mathbb{N} \times \mathbb{R}^+$ donde $\delta = 0$ si $l_i \in E$ y $0 < \delta \leq l_i$ en otro caso. Se llamará Pos_r al conjunto de posiciones de r. Para todo $i \geq 0$ el conjunto de posiciones de la forma (i, δ) caracteriza el conjunto de estados de Q por los que la ejecución r pasa mientras el tiempo fluye del estado q_i al q_{i+1}. Adicionalmente, si $q_i = (s_i, v_i)$ se notará $\Xi(i, \delta)$ al estado $(s_i, v_i + \delta)$.*

Si $r \in R_G$, se define a $\tau_r : \mathbb{N} \rightarrow \mathbb{R}^+$ como el tiempo transcurrido *en la ejecución r hasta alcanzar un estado dado de la ejecución partiendo de q_0 (estado inicial):*

$$\begin{array}{lcl}\tau_r(0) & = & 0 \\ \tau_r(i+1) & = & \left\{ \begin{array}{ll} \tau_r(i) & si\ l_i \in E \\ \tau_r(i) + l_i & si\ l_i \in \mathbb{R}^+ \end{array} \right.\end{array}$$

A su vez se define $\tau_r(i, \delta) = \tau_r(i) + \delta$.

1.3. Análisis utilizando regiones temporales

1.3.1. Discretización de estados

La lógica TCTL [ACD93] está asociada a los autómatas temporales; permite expresar propiedades verificables sobre éstos. Dado un autómata G y una propiedad de la lógica TCTL φ que predica sobre G, hacer *model checking* significa responder si $G \models \varphi$.

Un problema de los autómatas temporizados es que tienen una cantidad infinita de estados, dado que desde un estado se puede tomar una cantidad infinita de transiciones temporizadas[1]. Afortunadamente Alur, Courcoubetis y Dill consiguieron construir un grafo cociente finito a partir de un autómata G, y de tamaño exponencial con respecto a la cantidad de relojes involucrados. Esto se resume en el siguiente teorema:

Teorema 1.3.1 (El Model Checking por TCTL es PSPACE completo) *El model checking de* TCTL *sobre Autómatas Temporizados es PSPACE completo. Además, su complejidad temporal es de* $O(mn!2^n C^n)$ *donde* m *es el número de locaciones,* n *es el número de relojes, y* C *es el mínimo número natural de los que aparecen en las restricciones del Autómata Temporizado.*

Demostración

Ver [ACD93].

□

La idea intuitiva del teorema es que se puede partir el espacio (continuo) de valuaciones de relojes en una cantidad finita de clases de equivalencia.

Zonas y Regiones

Como mencionamos antes, usamos el concepto de *zona* para referirnos a un conjunto convexo (posiblemente infinito) de valuaciones de relojes; ésto se representa con una conjunción de desigualdades (por ejemplo: $(1 \leq x < 3) \wedge (2 < y \leq 5)$). Con *región* nos referimos a un conjunto (posiblemente infinito) no convexo de valuaciones, que interpretamos como una unión de zonas disjuntas.

1.3.2. Los operadores suc_t, suc_e y $suc_\triangleright$

A continuación se describe una familia de operadores que serán utilizados directamente en el cálculo de satisfacibilidad de un conjunto de requisitos por parte de un autómata temporizado. En primer lugar, se definirán operadores sobre valuaciones de relojes. Estos operadores se definen tomando como referencia a los que se hallan en [Sch02], donde se presentan operadores análogos, pero para el cálculo de predecesores.

Definición 1.3.1 (El operador suc_t) *Sea* $G =< S, X, E, A, I >$ *un autómata temporizado,* $s \in S$*,* $\psi \in \mathcal{P}(\mathcal{V}_X)$*,* $v \in \mathcal{V}_X$ *y* $\delta \in R^+$*. Se define el operador* $suc_t^s : \mathcal{P}(\mathcal{V}_X) \to \mathcal{P}(\mathcal{V}_X)$ *de la siguiente forma:*

$$(v + \delta) \in suc_t^s(\psi) \Leftrightarrow v \in \psi \wedge (s, v) \rightarrow^\delta (s, v + \delta)$$

Definición 1.3.2 (El operador suc_e) *Sea* $G =< S, X, E, A, I >$ *un autómata temporizado,* $s \in S$*,* $\psi \in \mathcal{P}(\mathcal{V}_X)$ *y* $v \in \mathcal{V}_X$*. Se define* $suc_e^s : \mathcal{P}(\mathcal{V}_X) \to \mathcal{P}(\mathcal{V}_X)$ *de la siguiente forma:*

[1] Esto es por tener variables de tiempo reales.

$$v \in suc_e^s(\psi) \quad \Leftrightarrow \quad (\exists < s, e, cond, asig, s' >, v')(\langle s, e, cond, asig, s' \rangle \in A \wedge v' \in \mathcal{V}_X \implies v' \in \psi \wedge v'[asig] = v) \wedge (s, v') \rightarrow^e (s', v)$$

Propiedad 1.3.1

Los operadores suc_t y suc_e son monótonos.

A continuación se introducen definiciones análogas a las recién presentadas, pero para operar sobre elementos de $\mathcal{P}(Q)$. Estas definiciones se derivan también de [Sch02].

En las siguientes dos definiciones, dado $\phi \in \mathcal{P}(Q)$, ϕ_s hace referencia a la restricción de ϕ a s; esto es, $(s', v) \in \phi_s \Leftrightarrow (s', v) \in \phi \wedge s' = s$.

Definición 1.3.3 (El operador suc_t sobre Q) *Sea $G =< S, X, E, A, I >$ un autómata temporizado, $(Q, \rightarrow)$ su sistema de transiciones asociado, $(s, v) \in Q$ y $\phi \in \mathcal{P}(Q)$. Se define $suc_t : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ de la siguiente forma:*

$$(s, v) \in suc_t(\phi) \Leftrightarrow v \in suc_t^s(\phi_s)$$

Definición 1.3.4 (El operador suc_e sobre Q) *Sea $G =< S, X, E, A, I >$ un autómata temporizado, $(Q, \rightarrow)$ su sistema de transiciones asociado, $(s, v) \in Q$ y $\phi \in \mathcal{P}(Q)$. Se define $suc_e : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ de la siguiente forma:*

$$(s, v) \in suc_e(\phi) \Leftrightarrow v \in \bigcup_{\langle s', e, cond, asig, s \rangle \in A} suc_e^{s'}(\phi_{s'})$$

Definición 1.3.5 (El operador $suc_\triangleright$ sobre Q) *Sea $G =< S, X, E, A, I >$ un autómata temporizado, $(Q, \rightarrow)$ su sistema de transiciones asociado, $(s, v) \in Q$ y $\phi \in \mathcal{P}(Q)$. Se define $suc_\triangleright : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ de la siguiente manera:*

$$(s, v) \in suc_\triangleright(\phi) \Leftrightarrow (s, v) \in suc_t(\phi \cup suc_e(\phi))$$

Dado que los operadores suc... devuelven conjuntos potencialmente infinitos de estados, y gracias al teorema 1.3.1, se pueden definir *estados simbólicos*:

Definición 1.3.6 (Estado simbólico) *Dado el autómata $G =< S, X, E, A, I >$, si $l \in S$ es una locación de G y z es una zona temporal, escrita como una restricción sobre relojes (Φ_X), (l, z) es un estado simbólico que representa a todos los estados $(l', \phi) \in Q$ tal que $l = l' \wedge \phi \models z$.*

De esa manera, el operador $suc_\triangleright$ sobre estados simbólicos tiene una definición análoga pero simbólica. Es decir, dado un estado simbólico (l, z), $suc_\triangleright(l, z)$ se computa realizando una manipulación simbólica (obviamente finita) sobre la representación de z.

La importancia de los estados simbólicos es que cada uno de ellos puede representar una cantidad infinita de estados de Q. Además, se puede realizar una manipulación simbólica y finita sobre su componente temporal para poder computar sus sucesores, testear inclusión, etc. El cálculo de sucesores se define brevemente en la siguiente definición:

Definición 1.3.7 ($suc_\triangleright$ sobre estados simbólicos) *$suc_\triangleright(l, z) = \{(l', z')/\langle s, e, cond, asig, s' \rangle \in E \wedge z' = suc_\tau(reset_\alpha(z \cap cond)) \cap I(s')\}$, donde $reset_\alpha$ significa poner los relojes en asig a cero y $suc_\tau(cond)$ significa reemplazar las restricciones de la forma $x \prec c$ por $x < \infty$ y dejando el resto intacto.*

Mas detalles pueden verse en [Sch07, Chapt. 2].

A partir de ahora, excepto que explícitamente se mencione, todos los estados son simbólicos.

1.3.3. Predicados sobre los sistemas de transiciones

Como se mencionó anteriormente, se cuenta con un formalismo lógico (la lógica temporal TCTL) para expresar propiedades sobre los autómatas temporizados. Sin embargo, gran parte de las propiedades que nos interesará verificar puede verse, más simplemente, como el problema de la *alcanzabilidad*. Informalmente, un estado será alcanzable desde un estado q (generalmente el inicial) si existe una ejecución perteneciente a $R_G(q)$ tal que dicho estado se encuentre en ella. Además, como parte del método de análisis, será necesario expresar propiedades sobre los estados alcanzables.

Para ello, para la versión simple de alcanzabilidad se define una función $Prop$ que mapea locaciones a conjuntos de variables proposicionales, y en este contexto:

- Las propiedades son variables proposicionales.
- Un estado simbólico (l, z) de Q satisface una variable proposicional $p \iff p \in Prop(l)$, y esto se nota $(l, z) \models p$.

1.3.4. El problema de la *alcanzabilidad*

El problema de la *alcanzabilidad* puede expresarse como:

> Dado un autómata temporizado G, un conjunto de requisitos R, ¿cuáles son aquellos estados desde los que existe una ejecución válida del autómata, de tal forma que dicha ejecución lo hace evolucionar a estados donde los requisitos expuestos en R valen?

Dentro del model checking, resolver el problema de la alcanzabilidad es crucial. Esto se debe a que muchas propiedades pueden ser expresadas como propiedades de alcanzabilidad.

A continuación describimos el método a seguir para resolver esta cuestión mediante los formalismos previamente presentados.

1.3.5. Estrategia de evaluación

Existen dentro de la bibliografía dos estrategias para abordar la cuestión de la alcanzabilidad. Son conceptualmente opuestas, en el sentido de que el recorrido que realizan sobre el sistema de transiciones es inverso.

En esta presentación se aborda una estrategia de evaluación *forward*. Como su nombre lo indica, esta estrategia consiste en recorrer el sistema de transiciones comenzando en los estados iniciales, intentando llegar a aquellos estados cuya alcanzabilidad se quiere verificar. Para esto se utiliza el operador $suc_{\triangleright}$.

Existe también otra estrategia, llamada *backwards*, la cual no se presentará en este trabajo. A modo introductorio, alcanza con decir que su enfoque es el inverso: intenta verificar la alcanzabilidad de un conjunto de estados comenzando el recorrido en éstos mismos, utilizando un operador llamado $pred_{\triangleright}$ (análogo a $suc_{\triangleright}$, pero tomando como base los predecesores en lugar de los sucesores). Utilizando este operador, se intenta llegar a los estados iniciales del sistema. El método para la verificación se basa, a su vez, en un cálculo de punto fijo. Los detalles de este enfoque pueden encontrarse en [Yov96].

A continuación presentamos el procedimiento básico para la exploración *forward* del sistema, junto a algunas notas para su mejor comprensión.

La idea del algoritmo 1.3.1 consiste en explorar el espacio completo de posibilidades, chequeando en cada paso si el estado que se ha encontrado (mediante el operador $suc_{\triangleright}$) es realmente nuevo o si, en algún otro momento de la ejecución, ya se ha pasado por dicho estado. La función del contenedor

1: **function** AlcanzabilidadForward(Property ϕ) $\gg$ ϕ: predicado $\in$ Pred(Q) 1.3.3)
$\gg$ Devuelve si es posible alcanzar un estado que satisfaga ϕ
2: $Visited \leftarrow \emptyset$
3: $Pending \leftarrow \{(l_0, R_0)\}$ $\gg$ l_0 es la locación inicial, y R_0 la zona que representa a la valuación inicial
4: **while** $Pending \neq \emptyset$ **do**
5: $(l, R) \leftarrow$ next(Pending)
6: **if** $(l, R) \models \phi$ **then** $\gg$ Ver (1.3.3)
7: **return** YES
8: **end if**
9: **for** $(l', R') \in suc_{\triangleright}(l, R)$ **do**
10: $R_{l'} \leftarrow \bigcup_{(l,z)\in(Visited_{l'} \cup Pending_{l'})} z$
11: **if** $R' \not\subseteq R_{l'}$ **then**
12: Add($(l', R'), Pending$)
13: Add($(l', R'), Visited$)
14: **end if**
15: **end for**
16: **end while**
17: **return** NO
18: **end function**

Algorithm 1.3.1: Algoritmo de alcanzabilidad Forward.

Pending es almacenar los estados que aún no han sido visitados, mientras que la del contenedor *Visited* es, análogamente al anterior, almacenar aquellos que sí se han recorrido. Además, $Visited_{l'}$ y $Pending_{l'}$ son las respectivas proyecciones de estos conjuntos que contienen sólo los estados cuya locación es l'.

Una característica problemática de este algoritmo (que lo hace extremadamente costoso) es la necesidad de chequear si cierto estado pertenece o no al conjunto de estados ya visitado; como en aquellas operaciones que se llevan a cabo en las líneas (10) y (11).

Acerca de las operaciones $\bigcup$ (paso 10) y $\not\subseteq$ (paso 11), pueden interpretarse intuitivamente como la unión y la relación *no contenido en* entre conjuntos. Al igual que en el caso de $suc_{\triangleright}$, estas operaciones de unión e inclusión tienen su versión simbólica que permite resolver la operación en una cantidad finita de pasos. Para más información al respecto ver [Pav06].

Capítulo 2

Antecedentes bibliográficos

Dada la complejidad a veces privativa de verificar propiedades en modelos temporizados, los trabajos realizados en el campo del model checking han tenido en gran parte un denominador común: sea mediante la distribución del trabajo en varias unidades de cómputo, optimización de algoritmos [BPO03] y estructuras, o reducción del sistema a verificar [BGO04b, Daw98], siempre se busca atacar el problema de la explosión combinatoria de estados. Esta explosión es producto de la composición de los distintos autómatas que representan el sistema. Por otra parte, ya vimos que la discretización del sistema genera una cantidad inmensa de regiones de equivalencia.

En los model checkers temporizados existentes que han alcanzado cierto grado de éxito, se han utilizado casi con exclusividad las DBMs (del inglés *Difference Bound Matrices*, matrices de cotas de diferencias) para representar las clases de equivalencia de los valores de los relojes. Estas estructuras están presentadas en detalle en [Dil90], por lo que nos limitaremos a dar una introducción que nos permita referirnos a ellas con comodidad en el resto del trabajo.

2.1. DBMs

Definición 2.1.1 (Cotas) *Una cota es un par* $(c, \prec)$*, donde* $c \in \mathbb{Z} \cup \{\infty\}$ *(entendiendo a* ∞ *como una constante más) y* $\prec \in \{<, \leq\}$*. Se define también el orden en* $\mathbb{Z} \cup \{\infty\}$ *extendiendo el orden en* $\mathbb{Z}$ *de forma que para todo* $c \in \mathbb{Z}$*, valga que* $c < \infty$*. Asimismo, se define el orden en* $\{<, \leq\}$ *de forma que* $<$ *es menor que* $\leq$*. Además, dada una cota* $C = (c, \prec)$*, notaremos* $-C$ *a la cota* $(-c, \prec)$*.*

Definición 2.1.2 (Orden total entre cotas) *Se dice que una cota* $(c, \prec)$ *es más estricta que otra* $(c', \prec')$*—notado* $(c, \prec) < (c', \prec')$*—si* $c < c'$ *o bien* $c = c'$ *y* $\prec < \prec'$*.*

Definición 2.1.3 (DBM) *Una DBM de dimensión* n *es una matriz cuadrada de* $(n+1) \times (n+1)$*, cuyos elementos son cotas.*

La idea detrás de esta estructura es, dada una DBM M, que M_{ij} representa la cota superior de la diferencia de relojes $x_i - x_j$. Por ejemplo, $x_i - x_j \leq 8$ se representará guardando en M_{ij} la cota $(8, \leq)$, mientras que $x_i - x_j > 2$ se codificará con la cota $(-2, <)$ en M_{ji}. Mediante esta codificación, todas las posibles DBMs representan el subconjunto de los poliedros convexos[1] que pueden denotar las soluciones de una restricción temporal. Por ejemplo, la restricción $x_1 \leq 2 \wedge x_2 > 3$ se ve representada, con una DBM y geométricamente, en la figura 2.1.

[1] Informalmente, se llama *convexos* a los poliedros (abiertos o cerrados) tales que dados dos puntos cualesquiera del mismo, el segmento que los une se encuentra íntegramente dentro del poliedro.

$$\begin{array}{c} \\ x_0 \\ x_1 \\ x_2 \end{array} \begin{array}{c} \begin{array}{ccc} x_0 & x_1 & x_2 \end{array} \\ \begin{pmatrix} (0,\leq) & (\infty,<) & (-3,<) \\ (2,\leq) & (0,\leq) & (\infty,<) \\ (\infty,<) & (\infty,<) & (0,\leq) \end{pmatrix} \end{array}$$

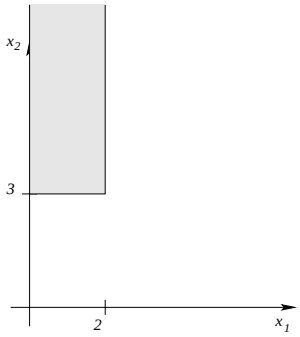


Figura 2.1: Una DBM y la zona representada.

Las DBMs no sólo tienen utilidad para representar las zonas temporales, sino que además las operaciones sobre estas zonas se traducen directamente como operaciones simbólicas sobre las DBMs. Además, cumplen con ciertas propiedades que las hacen aún más útiles para la verificación temporizada. A continuación mencionamos las operaciones y propiedades más importantes, dejando al lector la posibilidad de ahondar en ellas en [Dil90].

Observación 2.1.1 *Si bien existe más de una manera de representar una zona sobre una DBM, puede definirse una* forma canónica *tal que dos DBMs M y M' representan la misma zona si y sólo si son idénticas. Esto reduce la verificación de igualdad entre DBMs a un chequeo prácticamente sintáctico.*

La observación 2.1.1 ofrece una forma fácil de verificar vacuidad (o universalidad) de la zona representada por la DBM; de otra manera, sería necesario resolver las correspondientes restricciones temporales y verificar su equivalencia. Por otra parte, es también útil para la mayoría de las operaciones, ya que requerirán que sus parámetros se hallen en forma canónica. A continuación detallamos cómo puede transformarse una DBM cualquiera a su equivalente en forma canónica.

Definición 2.1.4 (Obtención de forma canónica de una DBM) *Dada cualquier DBM M, es posible obtener la forma canónica de la misma. Informalmente, esta forma canónica es tal que ninguna de las cotas puede ajustarse*[2] *más sin alterar el poliedro original.*

En la práctica, se puede obtener la forma canónica de una DBM representándola como un grafo con ejes pesados, y aplicando algoritmos de obtención de caminos mínimos sobre la misma; por ejemplo, el algoritmo de Floyd-Warshall. Si n es la cantidad de relojes presentes en el sistema, obtener la forma canónica de una DBM es $O(n^3)$, por lo que se intenta mantener la forma canónica durante la ejecución de las demás operaciones, ya que canonizar resulta una operación costosa.

Operaciones sobre DBMs

A continuación definiremos las operaciones más salientes a realizar con las DBMs; aquellas que serán necesarias para llevar a cabo la verificación temporizada.

Definición 2.1.5 (Intersección) *Dadas dos DBMs M y M' de dimensión n, se define la DBM intersección de ambas, I, también de dimensión n, de la siguiente forma, para todo $0 \leq i, j \leq n, i \neq j$:*

$$I_{ij} = \text{mín}(M_{ij}, M'_{ij})$$

donde mín *denota el mínimo según el orden presentado en la definición 2.1.2.*

[2] Se entiende por ajustar una cota al hecho de tornarla más estricta.

Definición 2.1.6 (Chequeo de inclusión) *Dadas dos DBMs M y M' de dimensión n, y tales que ambas se encuentran en forma canónica, puede determinarse si M representa una zona incluida en M' mediante la siguiente definición:*

$$M \subseteq M' \Leftrightarrow \forall 0 \leq i, j \leq n, M_{ij} \leq M'_{ij}$$

Observación 2.1.2 *En realidad, para asegurar la correcta operación de chequeo de inclusión de la DBM M en M', sólo es necesario que M' se encuentre en forma canónica, y no M [BY]. Esta observación cobrará mayor importancia más adelante cuando presentemos las RDBMs.*

Definición 2.1.7 (Reseteo de relojes) *Dada una DBM M se define la DBM R, que se obtiene como representación del reseteo de los relojes de un conjunto X, de la siguiente manera, para todo $0 \leq i, j \leq n$:*

$$R_{ij} = \begin{cases} (0, \leq) & si\ (x_i \in X \wedge x_j = 0) \vee (x_j \in X \wedge x_i = 0) \vee (x_i \in X \wedge x_j \in X) \\ M_{0j} & si\ x_i \in X \wedge x_j \notin X \\ -M_{i0} & si\ x_j \in X \wedge x_i \notin X \\ M_{ij} & en\ otro\ caso \end{cases}$$

Puede verse fácilmente que la operación presentada en la definición 2.1.7 es en realidad una proyección del poliedro sobre los ejes que representan los relojes que se resetean.

Definición 2.1.8 (Avance del tiempo) *Dada una DBM M, se define la DBM A, que representará las valuaciones de los relojes al avanzar el tiempo, de la siguiente manera, para todo $0 \leq i, j \leq n$:*

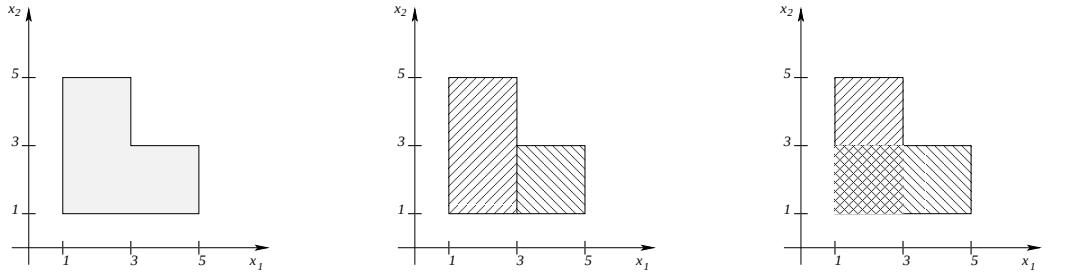
$$A_{ij} = \begin{cases} (\infty, <) & si\ j = 0 \\ (-\infty, <) & si\ i = 0 \\ M_{ij} & en\ otro\ caso \end{cases}$$

2.1.1. RDBMs

Las RDBMs (por *Reduced* DBMs, DBMs reducidas), descriptas entre otros en [BY], son una manera alternativa de representar las zonas. La idea es idéntica a la de las DBMs; sin embargo, el espacio que consumen es menor. La motivación detrás de la reducción es que, en general, resulta innecesario almacenar *todas* las diferencias entre los relojes, ya que varias de ellas se deducen a partir de las demás. El procedimiento de reducción consiste, como en el caso de la obtención de la forma canónica, en encontrar las cotas redundantes mediante algoritmos de determinación de caminos mínimos.

La representación obtenida mediante este proceso efectivamente anula las cotas innecesarias. Sin embargo, dado que podrían existir varios subconjuntos minimales de cotas que describan la misma DBM, la representación de las RDBMs no es canónica. Esto, en general, no es un problema ya que la mayoría de las operaciones que describimos anteriormente para DBMs pueden trasladarse a RDBMs sin cambios, ya que no exigen que las DBMs se hallen en forma canónica. La excepción a esta apreciación es la operación de inclusión, pero la observación 2.1.2 ofrece un resultado que permite, de todas maneras, obtener una reducción del tiempo requerido también para esta operación.

Más allá de la reducción del tamaño, los espacios que pueden representarse tanto con DBMs como con RDBMs son los mismos. Por otra parte, si bien ambas estructuras pueden representar sólo cierto subconjunto de poliedros convexos, suelen utilizarse conjuntos de DBMs (o RDBMs) para representar espacios no convexos. Evidentemente, tampoco existe una forma canónica de representar este espacio: múltiples conjuntos de poliedros convexos pueden representar, unidos, el mismo poliedro no convexo. Por ejemplo, el poliedro de la figura 2.2a puede representarse mediante esta noción conjuntista de varias formas: las figuras 2.2b y 2.2c son posibles representaciones. Puede notarse en la figura 2.2c que los conjuntos no son necesariamente disjuntos. Este tipo de problemas, como veremos luego, es el punto que se ataca utilizando el resto de las estructuras.



(a) Poliedro no convexo (b) Posible representación con uniones de convexos (c) Otra posible representación

Figura 2.2: Poliedro no convexo y posibles representaciones mediante convexos

2.2. Diagramas Binarios de Decisión

Los Diagramas Binarios de Decisión (BDDs, por el inglés *Binary Decision Diagrams*), son las estructuras en las cuales se fundan los *e*CDDs, los cuales nos proponemos optimizar. Estas estructuras no fueron desarrolladas con el model checking temporizado en mente, sino que fueron presentadas en [Bry86] como una forma de representar funciones booleanas (es decir, aquellas donde sus variables y su resultado son booleanas).

Definición 2.2.1 (BDD) *Un BDD es un grafo dirigido, con conjunto de vértices V, con un nodo distinguido llamado* raíz*, Cada vértice $v \in V$ puede ser* terminal *o* no terminal*. Los vértices no terminales poseen un índice—$indice(v) \in \{1..n\}$—y dos hijos $cero(v)$ y $uno(v)$, tal que ambos pertenecen a V. Los vértices terminales tienen un valor—$valor(v) \in \{0, 1\}$.*

Notaremos, cuando el contexto lo permita, tan sólo v para denotar el BDD cuya raíz es v.

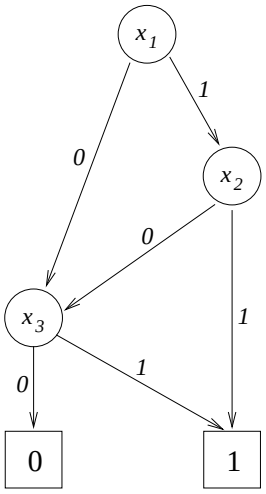


Figura 2.3: Un BDD

Intuitivamente, los vértices no terminales de un BDD representan las variables de la función booleana, mientras que sus hijos *cero* y *uno* representan las funciones que se obtienen fijando sus variables, justamente, en 0 y 1 respectivamente. Realizando este proceso de manera sucesiva, se alcanza algún vértice terminal, que representará o bien la función 0, o bien 1. Vale notar que dada cierta asignación a las variables, el camino a recorrer es único. Además, no existen nodos del BDD

que sean inalcanzables: siempre existe una asignación inicial que resulta en un camino que llega a un nodo dado. Por ejemplo, el BDD que se exhibe en la figura 2.3 es la representación de la función booleana $(x_1 \wedge x_2) \vee x_3$.

Sin embargo, los BDDs alcanzan mayor potencial al *reducirlos*. Avanzaremos con unas definiciones y propiedades que nos permitirán ver esta particularidad. No incluiremos las demostraciones, las cuales pueden ser consultadas en la fuente citada anteriormente.

Definición 2.2.2 (Isomorfismo de BDDs) *Dos BDDs B y B', con conjuntos de vertices V y V' respectivamente, se dicen* isomorfos *si existe una función biyectiva $\sigma : V \to V'$ tal que para todo $v \in V$, si $\sigma(v) = v'$, se cumple que:*

- *o bien tanto v como v' son terminales y $valor(v) = valor(v')$;*
- *o bien ambos son no terminales, y tales que $indice(v) = indice(v')$, $\sigma(cero(v)) = \sigma(cero(v'))$ y $\sigma(uno(v)) = \sigma(uno(v'))$.*

Definición 2.2.3 (RBDD (BDD reducido)) *Un BDD se dice* reducido*—y se nota* RBDD*, por el inglés* reduced *BDD—si no tiene ningún vértice v tal que $cero(v) = uno(v)$, ni tampoco posee dos vértices v, v' tales que $v \neq v'$ y además sean isomorfos.*

De esta manera, puede verse que dada una función booleana f y un orden inicial entre las variables, existe un único RBDD que la representa (exceptuando isomorfismos). En la literatura, suele llamarse *ROBDD*—BDD reducido y ordenado, por el inglés *Reduced Ordered BDD*—a los RBDDs que respetan un orden entre sus variables.

Este orden de variables no es un factor menor; el tamaño que puede llegar a desarrollar un RBDD depende fuertemente del ordenamiento de sus variables (en el sentido en que se encuentran tomando el recorrido desde la raíz). Como puede verse en la figura 2.4, ambos RBDDs representan la misma función booleana, $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$, pero el ordenamiento resulta crítico en el tamaño del grafo. En general, deben estudiarse la naturaleza y estructura del problema para determinar el ordenamiento de las variables que producirán el RBDD más compacto.

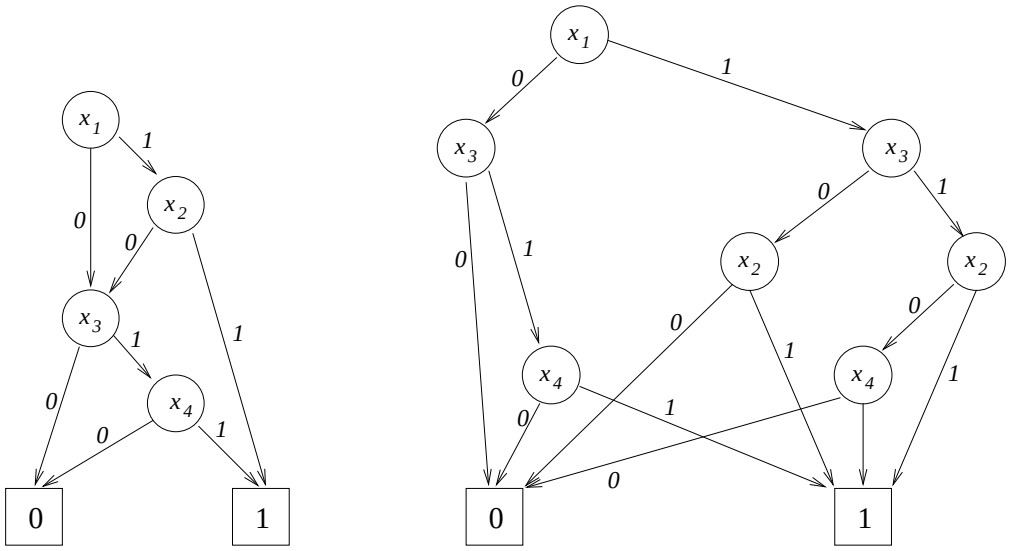


Figura 2.4: RBDDs con distinto orden

En la última década, varios trabajos han avanzado con el objetivo de generalizar los resultados obtenidos mediante los RBDDs en otras áreas, intentando integrarlos en las herramientas de model

checking como una forma de representar las regiones temporales. La idea detrás de este enfoque es intentar representarlas de manera compacta. Esto contrasta con la representación mediante DBMs, donde las regiones simplemente se representan mediante un conjunto de zonas. Esta representación no es óptima, ya que este conjunto podría estar demasiado atomizado, o también ser muy redundante, con muchas zonas tales que sus intersecciones son no vacías. El objetivo buscado es obtener una representación tal que estos problemas sean minimizados.

De esta manera, se intenta reducir tanto el tamaño de las estructuras como el tiempo de ejecución necesario para arribar a un resultado. En esta dirección, en [Bal96] se realizó un primer intento de utilizar BDDs para codificar directamente la unión de las DBMs. Sin embargo, al utilizar esta representación el problema de calcular la unión de clases de equivalencia no se soluciona de manera exacta, sino de manera aproximada, mediante el uso de la cápsula convexa de la solución real. Esta aproximación es conservadora, dado que resulta en casos de falsos positivos para la alcanzabilidad, pero no cae en falsos negativos.

Como consecuencia de estos trabajos, el esfuerzo fue enfocándose cada vez más en el desarrollo de soluciones que permitiesen la representación discreta pero *no necesariamente de bifurcación binaria* de las distintas posibilidades de decisión. En este sentido es que surgen los Diagramas de Diferencias de Relojes (CDDs) [BLP+99], los Diagramas de Restricción de Relojes (CRDs) [Wan03] y los Diagramas de Codificación de Regiones (REDs) [Wan00], que presentaremos más adelante. En pasos intermedios, existieron otros trabajos de menor impacto, como los Diagramas de Decisión Algebraicos (ADDs, por el inglés *Algebraic Decision Diagrams*) [REC+93], y los Diagramas de Decisión Numéricos (NDDs, del inglés *Numerical Decision Diagrams*) [EMA+97]. Estas estructuras, si bien se plantearon con la posibilidad de su uso en el ámbito del model checking, sufren de diversas debilidades, principalmente debido a la naturaleza exclusivamente binaria de las decisiones en cada nodo.

El paso siguiente, entonces, fue desarrollar estructuras con el objetivo de representar el espacio denso de los valores de los relojes, ya no codificándolos mediante sus representaciones binarias, sino utilizando fuertemente el contexto de los problemas surgidos durante el recorrido de los sistemas temporizados. Como resultado de esta línea de trabajo aparecen los Diagramas de Decisión de Intervalos (IDDs, proveniente del inglés *Interval Decision Diagrams*) [KL98], y otras variantes, donde se comienza a abandonar las decisiones binarias, transformándolas en n-arias.

Como mencionamos anteriormente, otros han tomado la posta y, heredando las ideas de estas estructuras, hoy en día cuentan con mayor impulso otras alternativas que, sin embargo, carecen de validación independiente que pueda sostener (o dar por tierra) sus virtudes. A continuación presentaremos los CDDs, que fueron usados como base de la estructura que estudiamos en este trabajo.

2.3. Usando CDDs como estructura alternativa

2.3.1. Diagramas de Diferencia de Relojes

Los diagramas de diferencias de relojes (CDDs, del inglés *Clock Difference Diagrams*) [BLP+99] surgen como variante a los IDDs. En un CDD, los nodos del diagrama representan diferencias de relojes (a diferencia de los IDDs, que sólo representan a cada reloj), mientras que cada eje saliente de un nodo está etiquetado con un intervalo. De esta manera, el eje representa el camino a seguir en el diagrama si la diferencia de las valuaciones se encuentra comprendida en dicho intervalo, dado que los ejes salientes de un mismo nodo contienen intervalos disjuntos entre sí. Por ejemplo, la figura 2.5 ilustra un CDD junto con la región que éste representa; para no complicar la lectura del gráfico, se han omitido los intervalos cuyos ejes guían directamente al nodo $False$ (y se ha omitido también el mismo nodo $False$), aunque es válido aclarar que dicha omisión no existe en la definición de la estructura.

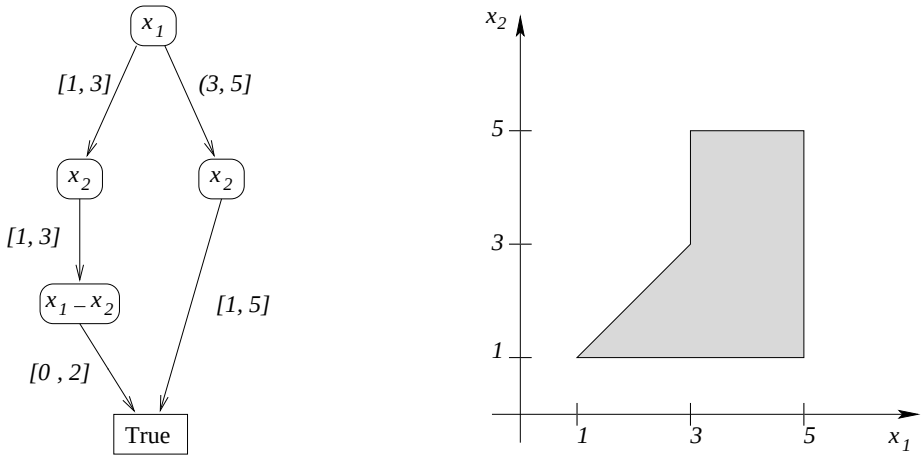


Figura 2.5: Un CDD y la región representada

En su presentación, los autores analizan también las posibilidades de compartir los subdiagramas isomorfos dentro del mismo diagrama, pero no exhiben detalles acerca de cómo hacerlo. Esto es importante, ya que es el principal argumento para sostener su bajo consumo de memoria. Además, presentan algunos algoritmos sobre esta estructura, todos ellos recursivos. Sin embargo, aparte de las cuestiones de eficiencia que surgen por el uso de recursión, estas operaciones sólo apuntan a utilizar la estructura como repositorio de regiones ya visitadas, es decir, como implementación del contenedor *visitados* del algoritmo 1.3.5, sin utilizarlas para el ciclo completo del model checking.

2.3.2. Extensión de los CDDs para representación de regiones

Tomando como base la definición inicial de CDDs, en [Pav06] se procedió a extender la estructura de datos (llamándola *e*CDD) para permitir representar regiones temporales de manera homogénea. De esta manera se pueden suprimir las (R)DBMs y usar solamente esta estructura para representar la unión de las zonas temporales encontradas. Para ser de utilidad, cada operación provista por las DBMs tiene definido su equivalente para *e*CDDs.

La implementación de *e*CDDs que pretendemos optimizar está diseñada para representar zonas con el menor desperdicio de espacio posible. En en el contexto un *e*CDD, una zona está representada por una cadena de nodos sin bifurcaciones, por lo que se empaquetaron estas cadenas de nodos de manera compacta. Naturalmente la estructura soporta ramificaciones para poder realizar uniones entre *e*CDDs.

2.3.3. Resultados del uso de eCDDs

Para compararlos con RDBMs, Los *e*CDDs se usaron de dos maneras:

- para reemplazar directamente cada RDBM por un *e*CDD, representando a cada región como a un conjunto de zonas representadas por *e*CDDs;
- para reemplazar a toda la secuencia de RDBMs por un sólo *e*CDD, utilizando la operación de unión entre *e*CDDs. Éste era el objetivo principal del desarrollo de *e*CDDs.

Usados como reemplazo 1 a 1 de los RDBMs, los *e*CDDs tuvieron muy buenos resultados, pues ésta estructura permite ahorrar canonizaciones en ciertos casos. De todas maneras, la cantidad de

zonas temporales no disminuyó, porque la representación de las regiones sigue siendo, en definitiva, una sumatoria de zonas.

El uso de un sólo eCDD para representar completamente a una región tenía el objetivo de detectar zonas que estuvieran incluídas dentro de una combinación del resto de las zonas ya agregadas a la región. En este sentido se consiguieron buenos resultados, dado que la cantidad de zonas temporales efectivamente decreció. La desventaja consiste en la cantidad prohibitiva de tiempo y memoria que consumen los eCDDs cuando se los usa como representación directa de una región. La degradación del tiempo de ejecución va de la mano del espacio consumido, por la gran cantidad de copias y posteriores eliminaciones de (sub)eCDDs. Quedó pendiente, entonces, investigar la hipótesis de que el mal comportamiento espacial de los eCDDs usados como regiones tuviera que ver con una gran redundancia interna; especialmente porque durante el algoritmo de unión se pueden duplicar subárboles en grandes cantidades.

Capítulo 3

Desarrollo de la Tesis

En este capítulo veremos, en primer lugar, el análisis llevado a cabo para cuantificar la redundancia interna de la estructura. Esto se realizó tomando mediciones sobre el *model checker* en funcionamiento.

Confirmada la existencia de redundancia, fue necesario diseñar un algoritmo de eliminación de redundancia de eCDDs, junto con las estructuras de datos auxiliares que fueran necesarias.

3.1. Análisis de la estructura de datos

La hipótesis inicial fue que el crecimiento explosivo de la estructura de datos se debía a una gran redundancia interna. El primer objetivo fue entonces cuantificar esta redundancia, tomando como modelo casos de la literatura. Para lograr esto se necesitó un método para identificar subárboles iguales. La igualdad entre eCDDs se definió, para este fin, en el sentido sintáctico.

3.1.1. Diferenciación de subárboles

En esta sección hablamos de "nodos". Estos son la representación física de un eCDD, y cuentan con el siguiente contenido básico:

- una lista de celdas, cada una de las cuales codifica una diferencia entre relojes y un intervalo;
- o bien un próximo nodo encadenado (el cual continúa la lista de celdas del ítem anterior), o bien una lista de tuplas (intervalo,eCDD hijo) correspondiente a una ramificación de la estructura.

Para identificar cada subárbol se implementó una función de *hash* que

- utilizara la información de cada celda de un nodo,
- tuviera en cuenta la estructura, incluyendo los *hashes* de los nodos hijos.

La función de hash estructural se definió en base a una función de hash de enteros de 64 bits [Wan97]. Aquí presentamos un pseudocódigo de la definición recursiva del hash de la estructura.

La definición del hash permite convenientemente calcularlo mientras se recorre la estructura, por lo que en lugar de implementarse la función de *hash* de manera explícita, se la integró a las operaciones ya existentes, de manera que cada una de éstas devuelve un eCDD con el hash correctamente calculado. Por tratarse de una función de hash, no se descarta la existencia de

```
1: function HashCelda(celda)                    ≫ Devuelve el hash correspondiente a una celda
2:     if celda = trueCell then
3:         return HASH_TRUE
4:     else if celda = falseCell then
5:         return HASH_FALSE
6:     else
7:         return hash(hash(celda.relojes) + hash(celda.minimo) + hash(celda.maximo))
8:     end if
9: end function

10: function HashCDD(cdd)  ≫ Devuelve el hash calculado a partir de este eCDD y todas sus
    celdas y CDDs hijos
11:     ret_value ← HASH_NULO
12:     for c ∈ cdd.celdas do
13:         ret_value ← hash(ret_value + HashCelda(c))
14:     end for
15:     for i ∈ cdd.intervalos do
16:         ret_value ← hash(ret_value + hash(i.minimo) + hash(i.maximo) +
    HashCDD(i.subcdd))
17:     end for
18:     return ret_value
19: end function
```

Algorithm 3.1.1: Algoritmos de cálculo de *hashes* para *e*CDDs

colisiones, por lo que dos *e*CDDs con el mismo hash podrían representar regiones diferentes. De todas maneras, sabemos que dos estructuras con distinto hash son al menos **sintácticamente distintas**[1].

3.1.2. Medición del grado de redundancia de la estructura

Teniendo un hash definido, fue posible tomar mediciones de la cantidad de hashes repetidos presentes en instancias de la estructura recorridas durante el funcionamiento del *model checker*.

Dado un *e*CDD X, llamamos X_r al espacio total ocupado por los sub-*e*CDDs de X repetidos, y sea X_t el espacio total ocupado por X. Podemos definir el porcentaje de desperdicio de un *e*CDD como la relación $\frac{X_r}{X_t}$.

Decidimos representar el tamaño de un *e*CDD X con la cantidad de celdas que contiene la estructura que lo implementa[2] (contando a las celdas de todos sus sub-*e*CDDs). Para calcular el espacio redundante fue necesario tener en cuenta la relación de parentesco entre *e*CDDs (para no contar varias veces el tamaño de un eCDD cuando las repeticiones son en realidad de un eCDD que lo contiene).

El comportamiento esperado (suponiendo que la redundancia sea un problema importante de la estructura) era que el porcentaje de desperdicio alcanzara un nivel significativo durante la ejecución del *model checker*.

[1] O dicho de otro modo, que su representación directa no coincide.

[2] Recordar que en la implementación de *e*CDD propuesta en [Pav06], las cadenas de *e*CDDs sin ramificar se almacenan en celdas contiguas.

Resultado de las primeras mediciones

Contrariamente a lo esperado, las primeras mediciones dieron como resultado un porcentaje de desperdicio que aumentaba y disminuía de manera oscilatoria. Esto podía tener varias causas:

- que la función de hash usada estuviera generando demasiadas colisiones;
- que la estructura misma cambiara demasiado durante la ejecución, de manera inherente a los algoritmos que operan sobre ella, lo cual podría invalidar el uso de hashes;
- que la estructura cambiara demasiado, pero por algún vicio de representación que pudiera ser salvado.

3.1.3. Investigación de la causa del desperdicio errático

Para descartar la primera opción se procedió a implementar una comparación completa entre *e*CDDs, y a usarla para comprobar si dos *e*CDDs con el mismo hash son efectivamente iguales. No se encontraron falsos positivos en los casos de test usados para las comparaciones realizadas en el capítulo de resultados (véase 5.1.1).

Acerca de las comparaciones entre e*CDDs.* La comparación utilizada en el algoritmo 3.2.1 es simplemente entre hashes (esto tiene sus implicaciones, ver nota que acompaña al algoritmo). De todas maneras se implementó una comparación sintáctica completa entre *e*CDDs para usarse en una etapa inicial de *debugging*. Esquemáticamente consiste en una recorrida *inorder* de ambos *e*CDDs a la par, y devolver *"son distintos"* a la primera diferencia observada entre los árboles. Esto se realiza, entonces, en un tiempo proporcional al tamaño del *e*CDD más pequeño de los dos.

Para discernir entre la segunda y tercera causa del problema, se implementó la estructura (y su operación de unión) en lenguaje funcional (Haskell) para poder hacer pruebas específicas fuera del contexto del *model checker*.

Se observó que, en ciertos casos, el algoritmo de unión generaba, para un nodo padre, intervalos contiguos[3] que contenían exactamente el mismo *e*CDD hijo. Estas secuencias de intervalos son semánticamente equivalentes a un sólo intervalo que los contuviera a todos juntos, y del que colgara una sola copia del *e*CDD hijo.

Esta repetición de intervalos es doblemente dañina, pues aparte del obvio desperdicio de espacio, se altera innecesariamente el hash de la estructura (que es puramente sintáctico). Una optimización simple consiste en detectar esta situación y fundir estos intervalos en uno solo. Se procedió a implementar esta mejora y a repetir las mediciones de desperdicio. Es importante aclarar que para detectar estos *e*CDDs iguales en intervalos contiguos, se hizo comparación directa de *hashes* en lugar de la comparación sintáctica mencionada en esta misma sección. Por lo tanto, esto tiene las mismas implicaciones que se comentan en 3.2.2 acerca del algoritmo 3.2.1.

Resultado de las segundas mediciones

Luego de la optimización mencionada en el apartado anterior, las mediciones reportaron un porcentaje de desperdicio de la estructura sin los altibajos observados anteriormente, lo cual sugirió que la eliminación de la redundancia de la estructura era viable. El porcentaje de redundancia observado en varios de los casos fue de al menos un 66 %.

[3] Por intervalos contiguos nos referimos a intervalos que coinciden en uno de sus extremos, y uno de estos es cerrado. Obsérvese que en este caso los dos intervalos son equivalentes a uno solo que los cubra completamente.

3.2. Diseño del método de eliminación de redundancia

En vista de los resultados obtenidos, se procedió a elegir una estrategia de detección y eliminación de la redundancia presente en la estructura. Dicha estrategia iría ligada a la operacion de unión entre dos eCDDs, que es la única que potencialmente agrega redundancia en la estructura.

3.2.1. Elección de la metodología de eliminación de redundancia

Se identificaron las siguientes aproximaciones al problema:

- eliminar un subárbol de manera *preventiva*; lo cual implica detectar (antes de empezar a unir dos sub-eCDDs) que el resultado de la unión ya existe en otra parte ya generada del árbol resultante,
- detectar que se acaba de generar un subárbol repetido, y proceder a eliminarlo, o
- realizar una segunda pasada por el árbol luego de unir, eliminando los subárboles redundantes encontrados.

Para los tres enfoques es necesario llevar un inventario de qué eCDDs fueron generados durante la unión.

El primer enfoque se puede atacar desarrollando una *cache* de operaciones de unión ya realizadas. El problema es que es necesario almacenar los operandos y resultados de todas las uniones realizadas hasta el momento (al menos en el contexto de una unión), de manera que el costo de búsqueda tenga poco impacto temporal, y que a la vez realice una eliminación de redundancia significativa.

El primer y segundo enfoque tienen el problema de que durante la unión es difícil saber si un subárbol generado en el resultado (en un instante dado del algoritmo) seguirá existiendo al final de la unión; el eCDD resultado se va ramificando una y otra vez a lo largo de la ejecución de la unión, lo cual agrega eCDDs hijos, elimina eCDDs previamente creados y modifica los hashes de sus eCDDs padres.

El tercer enfoque es el más simple de implementar. Sus principales desventajas son que es necesario recorrer nuevamente el eCDD resultante después de unir; y que, como elimina la redundancia al final, requiere que haya al menos memoria suficiente para soportar la generación de esa redundancia en primer lugar.

Luego de analizar los tres enfoques, se decidió por el tercero para el desarrollo de esta Tesis. Algunas razones de esta elección son:

- El método es simple de implementar.
- La implementación usando BDDs usa un método basado en la misma idea de comprimir después.
- La tasa de compresión de eCDDs conseguida con este método es la máxima conseguible haciendo *aliasing*, dado que se conoce a cada eCDD creado y por lo tanto se pueden eliminar todas las repeticiones. Esto nos da una cota superior de la ventaja de hacer *aliasing*.

3.2.2. El algoritmo top-down de eliminación de redundancia

El algoritmo de unión se modificó para que registrase en algún tipo de repositorio cada sub-eCDD nuevo creado, y el destructor de eCDDs se modificó para que eliminara del repositorio un eCDD *efectivamente* eliminado (puesto que un eCDD con un conteo de referencias mayor a 1 no

```
1: procedure CompressCDD(inout cdd, in tabla)          ≫ tabla: repositorio de sub-eCDDs
2:     if ya_comprimido(cdd) then
3:         return
4:     end if
5:     alias ← obtener_cdd(tabla, cdd.hash)
6:     if alias ≠ NULL and alias ≠ cdd then                              ≫ Ver nota al pie.
7:         destruir(cdd)          ≫ Existe reemplazante válido, se destruye el duplicado.
8:         cdd ← alias                                          ≫ Asignación de referencias
9:         if ya_comprimido(cdd) then
10:            return
11:        end if
12:    end if
13:    for i ∈ cdd.intervalos do
14:        CompressCDD(i.subcdd, tabla)
15:    end for
16:    marcar_cdd_comprimido(cdd)          ≫ Esto evita posibles recorridas repetidas
17:    return
18: end procedure
```

Algorithm 3.2.1: Algoritmo recursivo de eliminación de redundancia de eCDDs

será *efectivamente* eliminado). Luego de terminada la unión se procede a eliminar la redundancia con una recorrida *inorder*.

Comparación de e*CDDs.* En el paso 6, el término $alias \neq cdd$ corresponde a una comparación entre referencias (en tiempo constante). El *alias* obtenido de la tabla de eCDDs ya tiene el mismo *hash* que *cdd*, por lo tanto en este paso se está asumiendo que ambos eCDDs son iguales sólo basándose en la igualdad de *hashes*. Esta comparación de referencias se puede reemplazar por la comparación sintáctica mencionada en 3.1.3; esto agregaría un costo considerable a la compresión (pues debería realizarse antes de cada reemplazo). Como asimismo se comenta en 3.1.3, primero se comprobó que valerse sólo de la igualdad de hashes no fuera un problema en los casos de test usados para la comparación del capítulo de resultados.

Capítulo 4

Implementación del método de eliminación de redundancia

En este capítulo trataremos algunos detalles concernientes a la implementación de los algoritmos propuestos, y al diseño de una estructura de datos auxiliar que fue necesaria: la tabla de eCDDs (usada para registrar los sub-eCDDs creados durante la unión).

4.1. Revisión de la implementación de los eCDDs

Aquí resumimos los cambios necesarios realizados a la estructura para llevar a cabo la implementación del método de eliminación de redundancia.

- Se agregó un contador de referencias (necesario para implementar los mecanismos de *aliasing*).
- Se agregó un contador auxiliar de la cantidad de celdas que contiene un eCDD (incluídos sus eCDDs hijos), a efectos de tener una medida del tamaño de un eCDD en las mediciones.
- Se agregó una marca para indicar si el eCDD ya está comprimido al máximo, para ahorrar pasos innecesarios durante la compactación.
- Se agregó el campo *hash*, que almacena el resultado de aplicar al eCDD la función $HashCDD()$ ya definida. Ver además 4.2 y 4.3 para detalles de su uso.
- La operacion de destrucción de eCDDs fue adaptada para que tenga en cuenta el aliasing, y además no elimine de la tabla de eCDDs un eCDD que todavía tiene referencias.

Estos cambios fueron previos a la implementación del método de eliminación de la redundancia. La idea fue que las modificaciones sean compatibles con el algoritmo de unión original sin compresión; esto facilitaría la comparación del comportamiento del algoritmo con y sin eliminación de la redundancia. Estas modificaciones permiten que un eCDD comparta subestructuras de manera similar a un DAG[1].

4.2. Implementación del cálculo del hash de la estructura

La función de hash presentada en el capítulo anterior es de naturaleza recursiva. Implementarla de manera directa implicaría volver a recorrer toda la estructura para actualizar el hash.

[1] Directed Acyclic Graph, o Grafo Dirigido Acíclico

Las operaciones ya implementadas sobre *e*CDDs son de naturaleza iterativa, apilando subárboles para recorrer, copiar o generar la unión de *e*CDDs. La decisión tomada fue implementar el *hash* de la estructura de forma análoga. Pero hacer esto de manera directa sigue requiriendo recorrer la estructura nuevamente con el sólo propósito de calcular el hash de cada uno de sus subárboles.
En lugar de implementarla iterativamente en una función especial, entonces, se integró el cálculo del hash a:

- la copia de zonas,
- la copia de regiones,
- el cálculo de la unión entre una zona y una región.

De esta manera se aprovecha la recorrida de la estructura en las operaciones mencionadas, y se va calculando el *hash* sobre la marcha. Esto tuvo algunas complicaciones, dado que durante la copia los *e*CDDs se copian de manera *top-down*, y en la unión el resultado se genera de manera *bottom-up*. Hubo que tener en cuenta estas dos maneras diferentes de generar *e*CDDs para que no hubiera inconsistencias en los hashes generados.

4.3. Implementación de la tabla de *e*CDDs

Para implementar el algoritmo propuesto fue necesario diseñar e implementar una tabla que sirviera de *cache* de *e*CDDs generados durante una unión. Se consideraron las siguientes variables:

- tamaño (fijo o dinámico);
- complejidad de búsqueda, inserción y eliminación;

Algoritmos de inserción, eliminación y búsqueda

El principal objetivo fue que el uso de la tabla no encareciera demasiado el costo computacional del algoritmo de unión, por lo cual se optó por implementar la tabla como una *tabla de hash*. La misma es de *direccionamiento abierto*, haciendo *doble hashing* en caso de colisión. La función de hash es una versión de 32 bits de la misma función de hash utilizada para el hash de la estructura [Wan97].
Aún en caso de colisión, las iteraciones para inserción/búsqueda/eliminación están limitadas a una cantidad fija **N** para acotar la complejidad. Esto tiene sus consecuencias:

- si luego de **N** intentos no se encontró un lugar vacío para insertar un elemento, se eligió reemplazar alguno de los **N** anteriores para insertar el nuevo elemento;
- tanto la búsqueda como la eliminación pueden llegar a realizar **N** iteraciones si el elemento no estaba en la tabla.

Tamaño de la tabla

Al implementar la tabla, se eligió usar un tamaño de la forma 2^k. Al ser el tamaño potencia de 2, basta con tomar los últimos k bits del hash para tener la posición del *e*CDD en la tabla. En principio se fijó un tamaño arbitrario, pero esto tiene desventajas, ya que una tabla demasiado grande hace desperdiciar tiempo al vaciarla (lo cual se realiza antes de cada unión), y una demasiado pequeña puede no guardar suficientes *e*CDDs para hacer una eliminación eficiente de la redundancia.

Finalmente se decidió variar el tamaño dinámicamente para ajustar la tabla lo mejor posible durante la ejecución. La tabla se redimensiona cada 100 uniones, en caso de que el porcentaje

de ocupación de la tabla en el promedio de las 100 uniones está por debajo de 25 % del tamaño actual, o por encima del 85 % del mismo.

El cambio de tamaño es asimétrico. Tomando como tamaño inicial 2^k, la tabla se agrandará a 2^{k+2} (esto es, siempre se cuadruplica su tamaño). Puede hacer falta agrandar la tabla más de una vez en caso de que el tamaño actual esté muy desfasado del óptimo, pero es preferible ser conservador para agrandar la tabla, dado que ésto encarece la *performance*.

La tabla se achica de manera más agresiva. Si el tamaño actual es de 2^k, y el tamaño promedio observado fue de $j < 2^k$, se tomará como objetivo un exponente de $n = log2(j)$, y el tamaño final será de $2^{k-\frac{k-n}{2}}$. Por ejemplo, si $k = 20$ y $n = 10$, en un sólo redimensionamiento la tabla quedará de tamaño 2^{15}. Este abrupto decrecimiento hace que la tabla llegue rápido a un tamaño pequeño, lo que significa menos *overhead* en las uniones.

Almacenamiento de la tabla

La tabla de *e*CDDs se almacena en memoria principal, existiendo una única instancia global de ésta. La tabla se vacía antes de comenzar una nueva unión. Una posibilidad a considerar a futuro, es la de compartir *e*CDDs entre locaciones distintas, lo cual podría incrementar el porcentaje de *aliasing* conseguido.

4.4. Implementación del algoritmo top-down de eliminación de redundancia

Así como en el caso de los hashes, para la implementación de este algoritmo se procedió a convertirlo en un algoritmo de pila, para evitar las llamadas recursivas. Se utilizó la misma manera de recorrer *e*CDDs utilizada en otras funciones anteriormente implementadas.

Capítulo 5

Pruebas realizadas y resultados obtenidos

Con el objetivo de validar la implementación de aliasing de la estructura, comparar su desempeño contra el de otras implementaciones, y en última instancia, decidir si era justificable el uso de la técnica de *aliasing*, se procedió a medir de varias maneras la *performance* de cada implementación, utilizando para este fin varios casos de estudio tomados de la literatura.

5.1. Contexto de comparación

Plataforma de pruebas

Para todas las pruebas se utilizó un servidor FreeBSD 7.0 de 64 bits, procesador Intel Xeon de 8 núcleos a 1.6 GHz por núcleo[1], con 4 GB de memoria RAM disponible.

Variantes a comparar.

Para poner a prueba el desempeño del método de eliminación de la redundancia, se realizó una comparación entre varias versiones del *model checker*:

- la que emplea RDBMs (RDBM), tomada como base relativa para todas las mediciones de tiempo y espacio,
- la que emplea CDDs como reemplazo de RDBMs (zCDD),
- la que emplea CDDs para implementar regiones (usando la operación de unión) (eCDD),
- y por último, la anterior, pero con eliminación de la redundancia generada por la unión. (eCDD +a)

Variables de comparación.

De cada corrida de prueba se recolectaron los siguientes datos:

- El tiempo total (de reloj) transcurrido;

[1] La cantidad de núcleos presentes no es relevante en el rendimiento de las versiones del *model checker* estudiadas en este trabajo, al ser estas *single-threaded*. Por lo tanto, la velocidad de ejecución efectiva del *model checker* fue de 1.6 GHz para las pruebas realizadas.

- La cantidad máxima de memoria residente (esto es, física) usada por el proceso;
- La cantidad de locaciones encontradas durante la verificación;
- La cantidad de zonas procesadas[2];
- La cantidad total de chequeos de inclusión realizados;
- La cantidad de chequeos de inclusión exitosos.

5.1.1. Casos de prueba utilizados

Daremos aquí una breve descripción de los casos de prueba utilizados durante las pruebas.

FDDI4 a FDDI9

Esta familia de casos modela una extensión del protocolo de red token ring FDDI, en la cual un observador monitorea el tiempo que tarda un *token* en volver a una estación de la que fue liberado. Los tamaños de los casos utilizados son los siguientes:

- FDDI4: 157 locaciones, 9 autómatas, 14 relojes.
- FDDI5: 379 locaciones, 11 autómatas, 17 relojes.
- FDDI6: 881 locaciones, 13 autómatas, 20 relojes.
- FDDI7: 2021 locaciones, 15 autómatas, 23 relojes.
- FDDI8: 4141 locaciones, 17 autómatas, 26 relojes.
- FDDI9: 10097 locaciones, 19 autómatas, 29 relojes.

RCS4

RCS4, o *Railroad Crossing System* (Sistema de cruce de vías), presentado en [ACD+92], modela el comportamiento del sistema de una barrera de un paso a nivel de trenes, en la cual están presentes la barrera, el controlador de la barrera, y n trenes que pueden estar aproximándose o alejándose del paso a nivel. Un observador monitorea que la barrera siempre esté en la posición correcta (levantada o baja) según el estado de los trenes. El tamaño de este caso (de 4 trenes) es de 168 locaciones, 8 autómatas con un reloj cada uno.

Struct (ObsSliced)

Struct es un sistema de control de las vibraciones en una estructura que pueden ocurrir por vientos fuertes o terremotos. Esta variante fue preprocesada con el método de reducción de estados seguro *ObsSlice* [BGO04a]. El tamaño de este caso es de 129 locaciones, y cuenta con 7 autómatas de un reloj cada uno.

Conveyor6ab

Corresponde a *Conveyor Belt* [BOS05], el modelo de una cinta transportadora que tiene una cierta cantidad de puntos de detención, y otra cantidad de objetos recorriendo simultáneamente la cinta. Este caso cuenta con 6 puntos de detención y 2 objetos. Esto resulta en 643 locaciones, 11 componentes, sumando 10 relojes entre todos.

[2] La cantidad de zonas procesadas es equivalente a la cantidad de estados visitados en la verificación.

5.2. Comentario de los resultados

En las secciones siguientes presentamos los resultados obtenidos con cada caso de test. Los porcentajes de ahorro o desperdicio informados siempre serán comparando contra la versión RDBM, salvo que se indique lo contrario.

Independientemente de los resultados de tiempo y memoria obtenidos, es interesante destacar que la cantidad de chequeos de inclusión realizados durante la verificación disminuye considerablemente en las versiones con unión (eCDD y eCDD +a) en comparación con las versiones con RDBM y con eCDDs usados como zonas (zCDD). A medida que aumenta el tamaño de los casos de prueba, este ahorro crece. Podemos ver la tendencia en la siguiente tabla.

Caso	RDBM/zCDD	eCDD/eCDD +a	Diferencia
	Chequeos de inclusión		(Porcentaje)
FDDI4 True	977	753	-23 %
FDDI5 True	4121	2764	-33 %
FDDI6 True	15564	9186	-41 %
FDDI7 True	53223	28201	-48 %
FDDI8 True	131091	67847	-48 %
FDDI9 True	528134	229556	-57 %
RCS4 True	163740	5233	-97 %
Conveyor6ab True	26354	4969	-88 %
Struct True	3591	1072	-71 %

Cuadro 5.1: Chequeos de inclusión realizados por los diferentes *model checkers*.

La ventaja ganada con este ahorro de chequeos de inclusión se ve opacada por otros aspectos negativos de los eCDDs usados como regiones, como veremos a continuación.

5.2.1. FDDI4 y FDDI5

En ambos casos de prueba, el tiempo de ejecución es demasiado corto como para notar diferencias significativas.

5.2.2. FDDI6 a FDDI9

En esta serie de casos de pruebas puede verse una tendencia que se repite:

- zCDD es la que mejor se comporta en términos de tiempo (entre el 47 % y el 58 % de ahorro frente a RDBM) y espacio (mas del 50 % de ahorro de espacio, en los casos más grandes).
- eCDD es la que peor se comporta en espacio (hasta un 1172 % de incremento de espacio en FDDI9), pero es un 40 % más rápido.
- eCDD +a es levemente más lenta que eCDD (un 2 % en los casos más grandes), pero consume de un 30 % a un 40 % menos de memoria (con respecto a eCDD) gracias al aliasing.

5.2.3. RCS4

En este caso se ve claramente la diferencia entre eCDD y eCDD +a, pero ambos siguen siendo mucho más lentos que RDBM y zCDD. La primera optimización implementada explicada en 3.1.3 disminuyó dramáticamente el espacio y tiempo consumidos, y el agregado del *aliasing* lo mejoró un poco más.

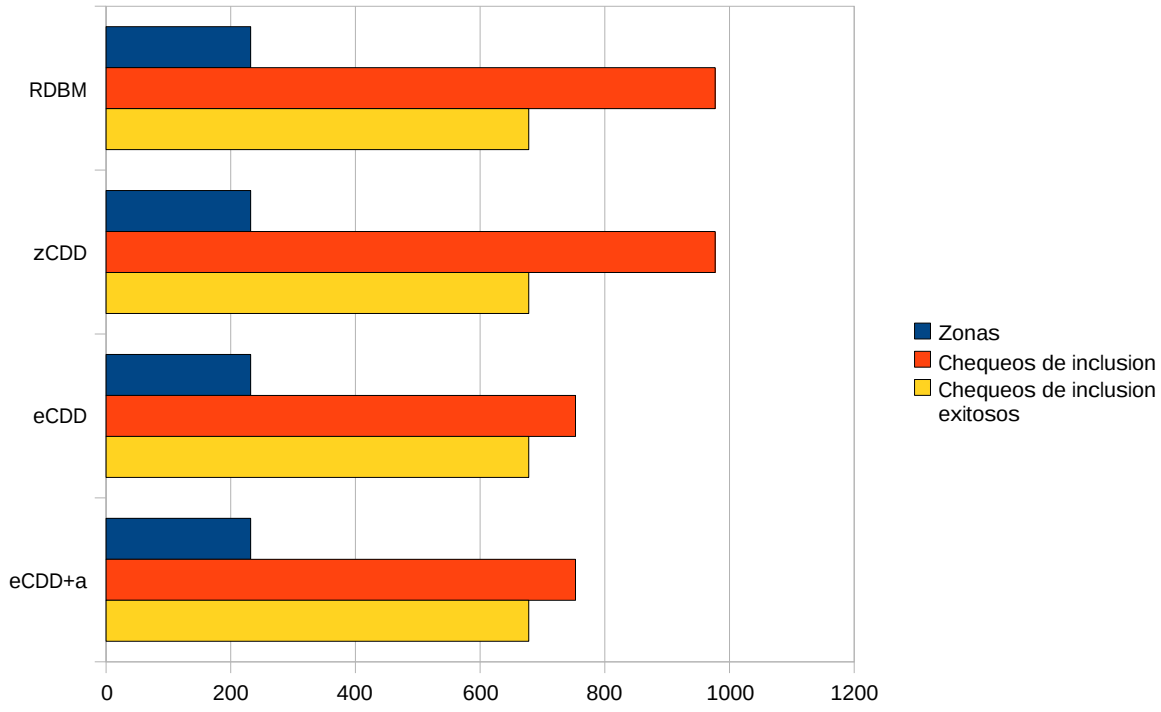


Figura 5.1: FDDI4 True: Zonas y Chequeos de inclusión

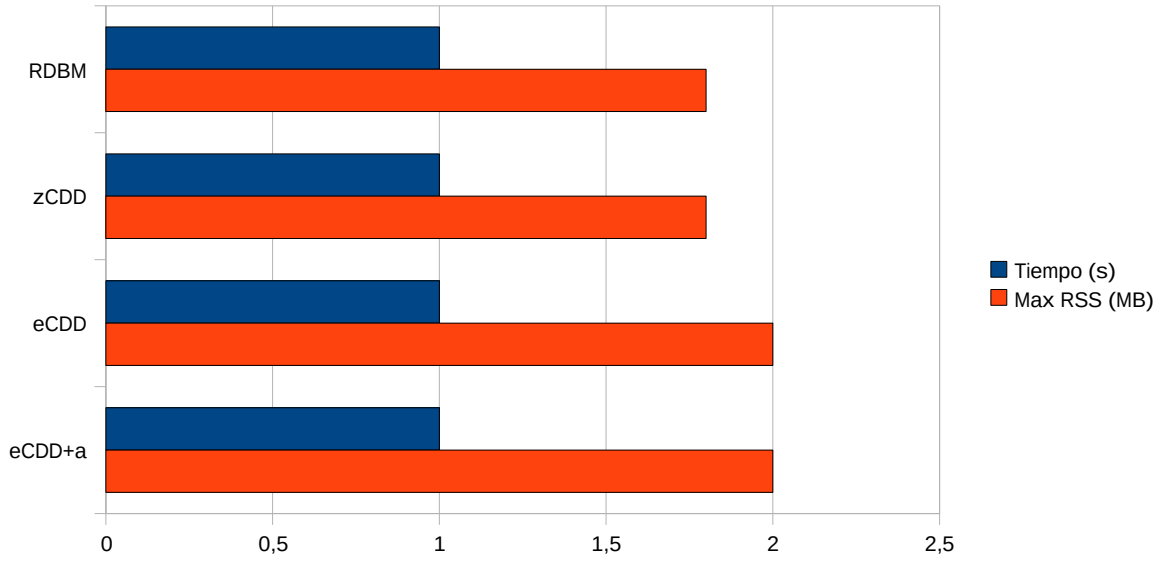


Figura 5.2: FDDI4 True: Tiempo y espacio

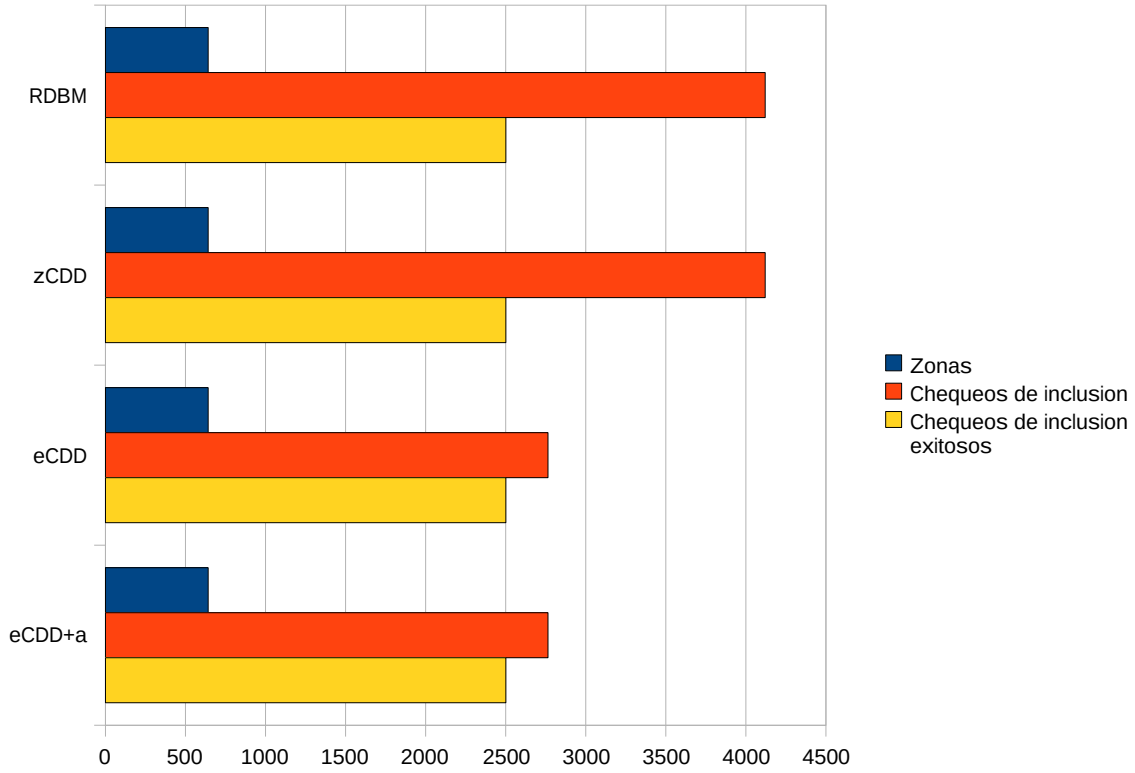


Figura 5.3: FDDI5 True: Zonas y Chequeos de inclusión

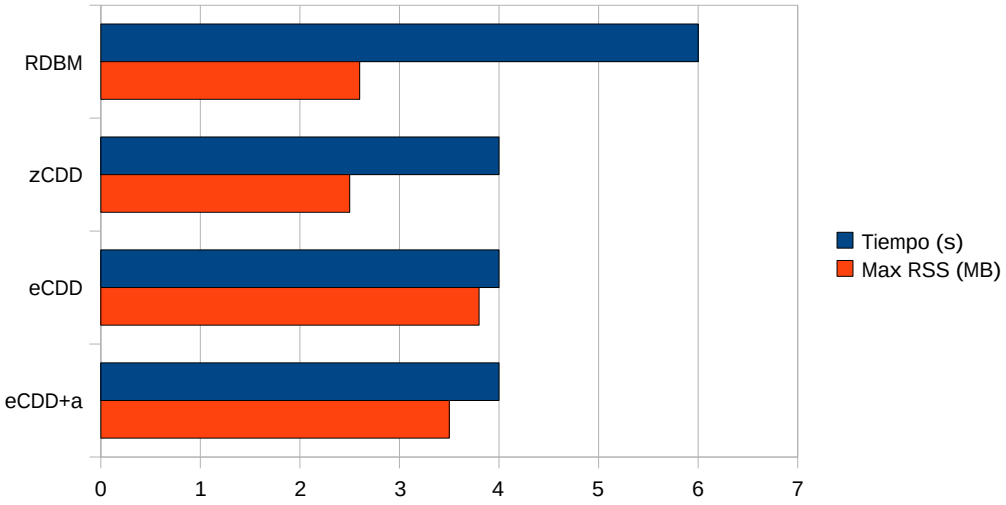


Figura 5.4: FDDI5 True: Tiempo y espacio

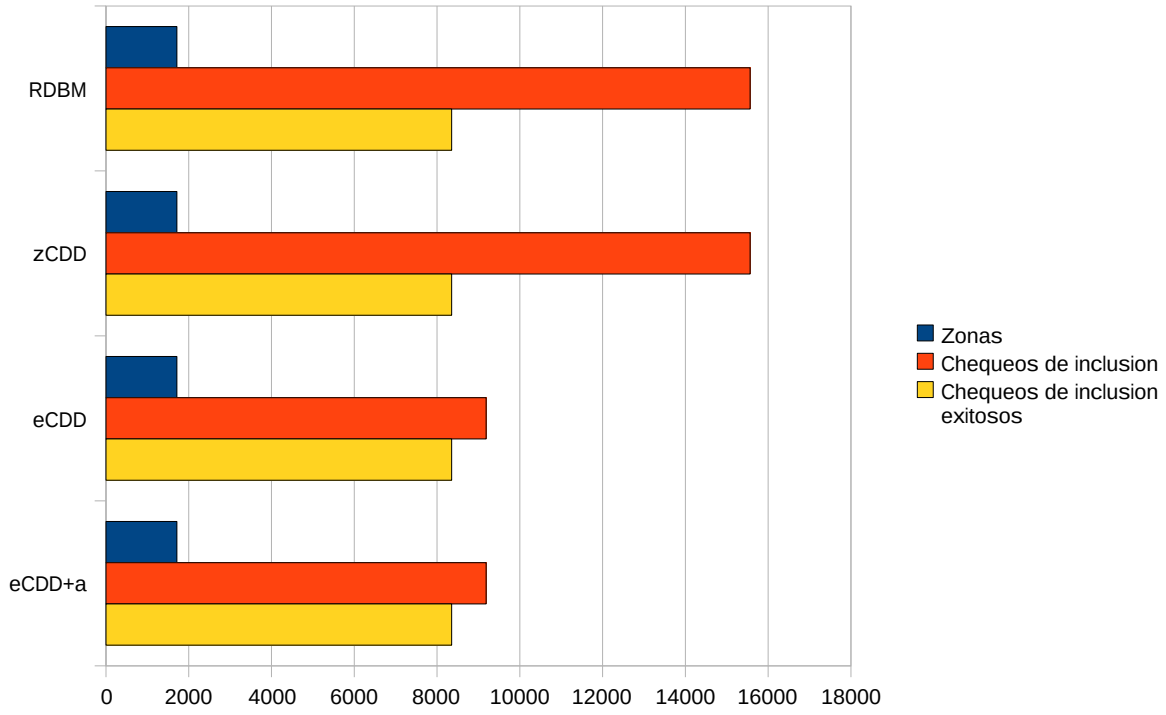


Figura 5.5: FDDI6 True: Zonas y Chequeos de inclusión

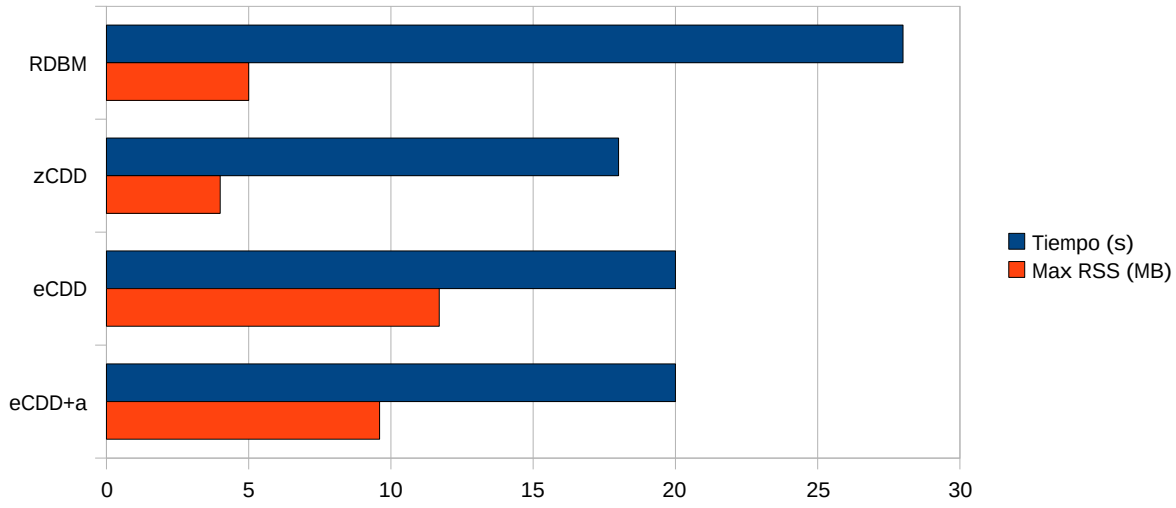


Figura 5.6: FDDI6 True: Tiempo y espacio

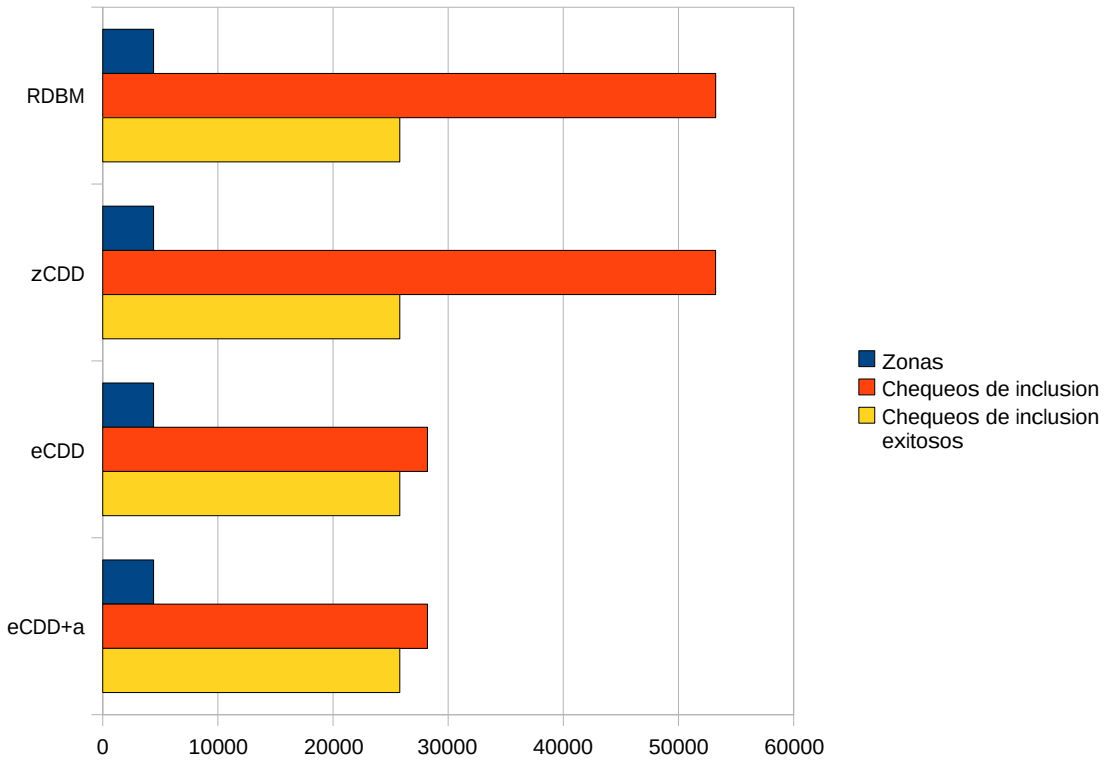


Figura 5.7: FDDI7 True: Zonas y Chequeos de inclusión

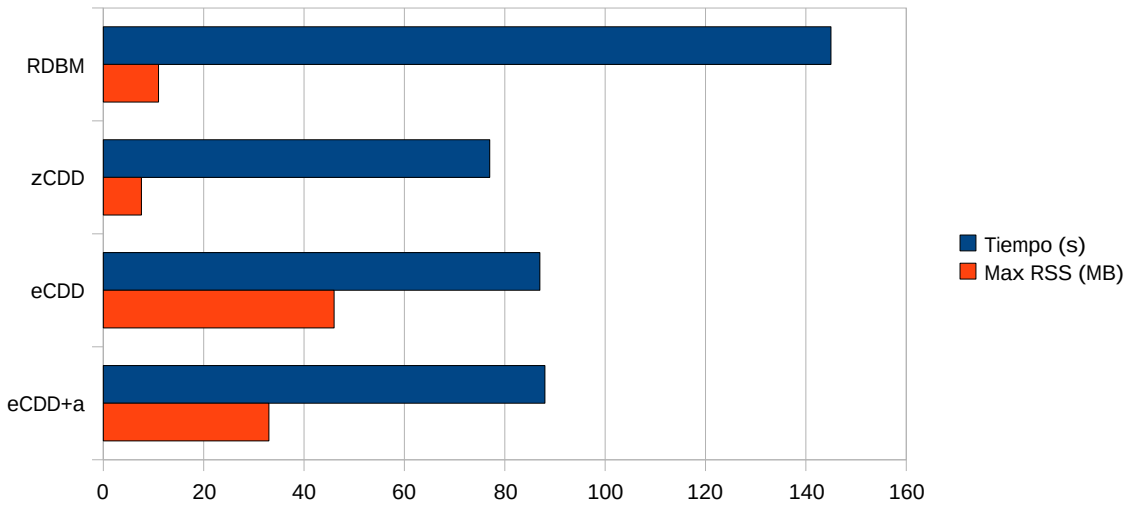


Figura 5.8: FDDI7 True: Tiempo y espacio

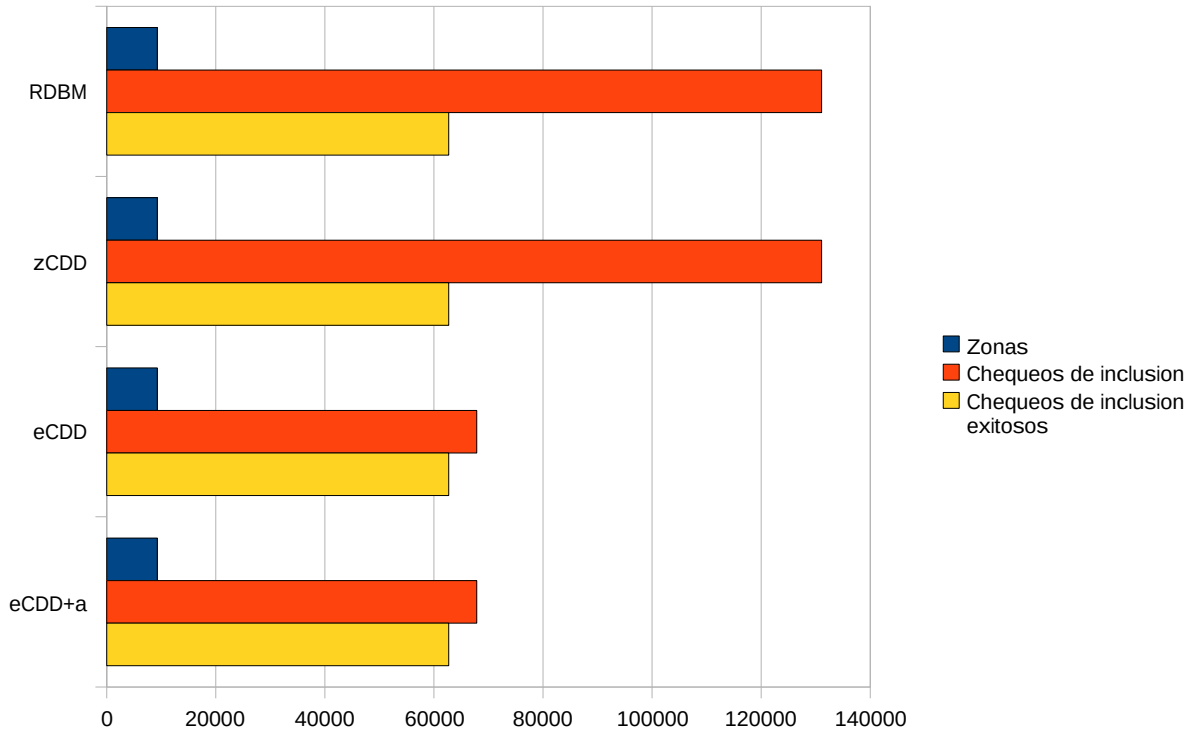


Figura 5.9: FDDI8 True : Zonas y Chequeos de inclusión

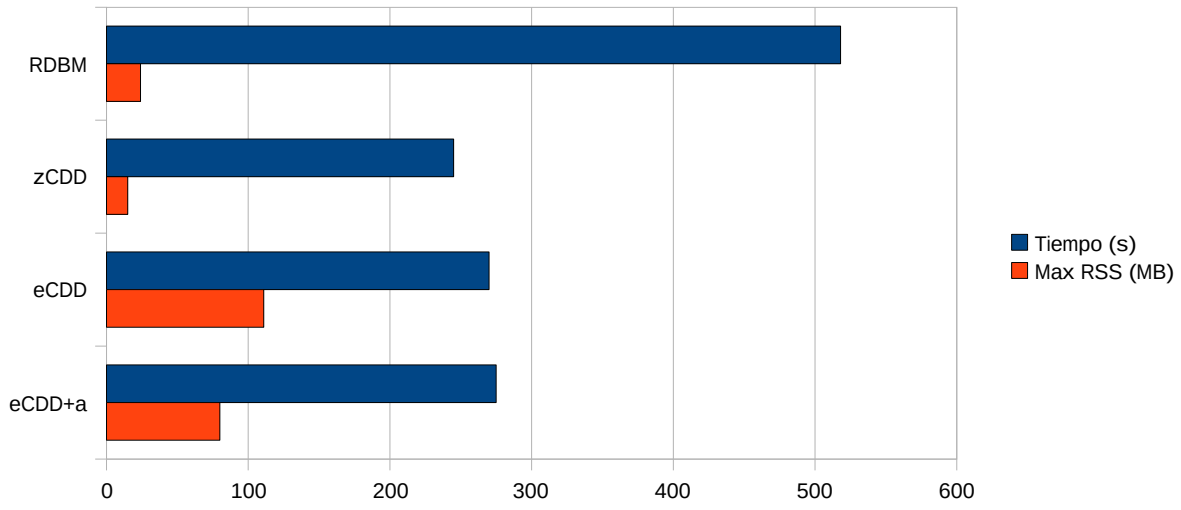


Figura 5.10: FDDI8 True: Tiempo y espacio

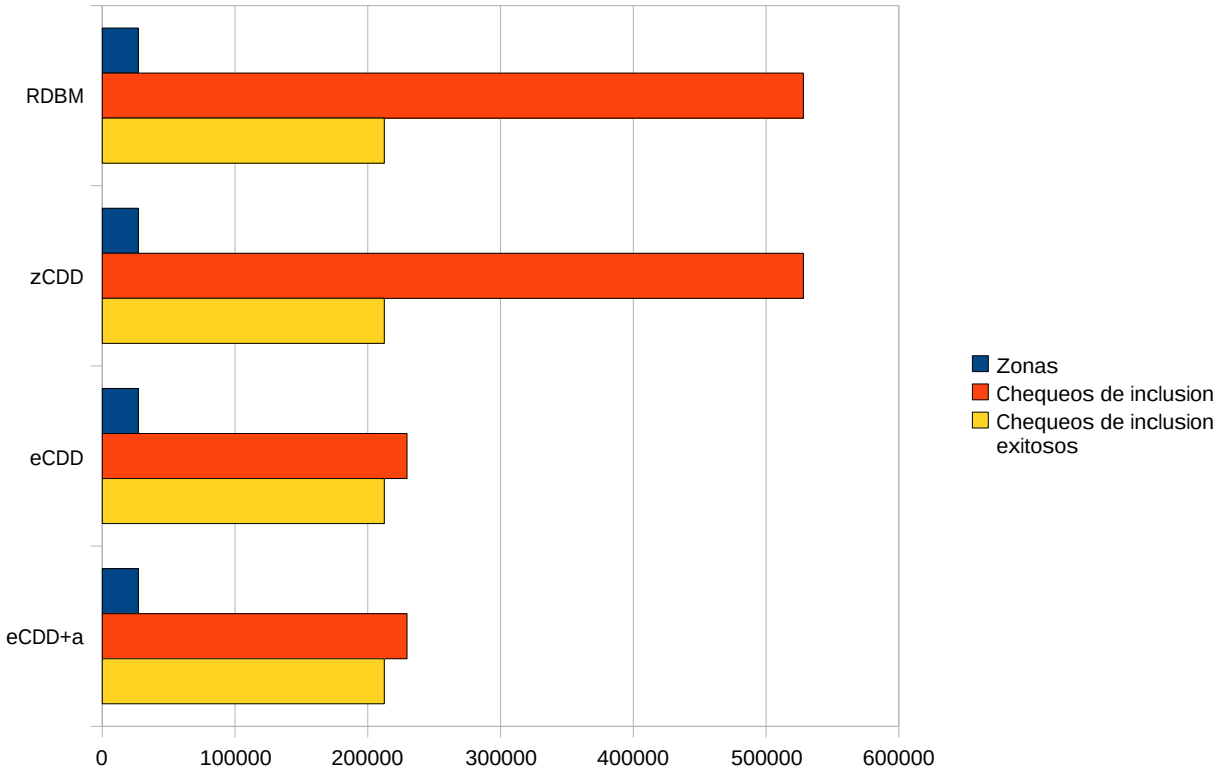


Figura 5.11: FDDI9 True : Zonas y Chequeos de inclusión

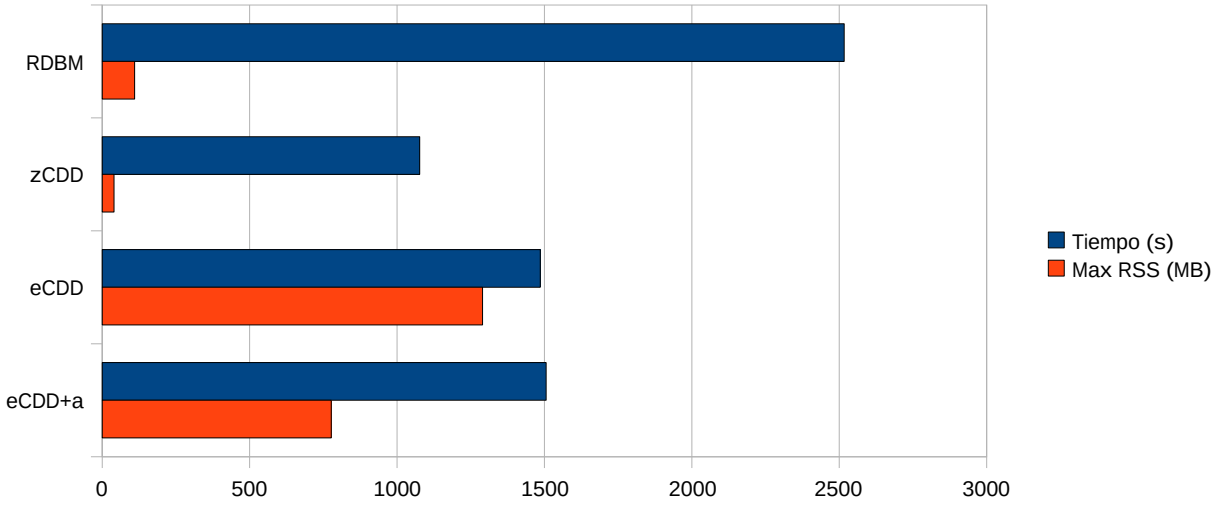


Figura 5.12: FDDI9 True: Tiempo y espacio

La ventaja de tiempo observada a favor de eCDD +a en comparación con eCDD se explica mediante

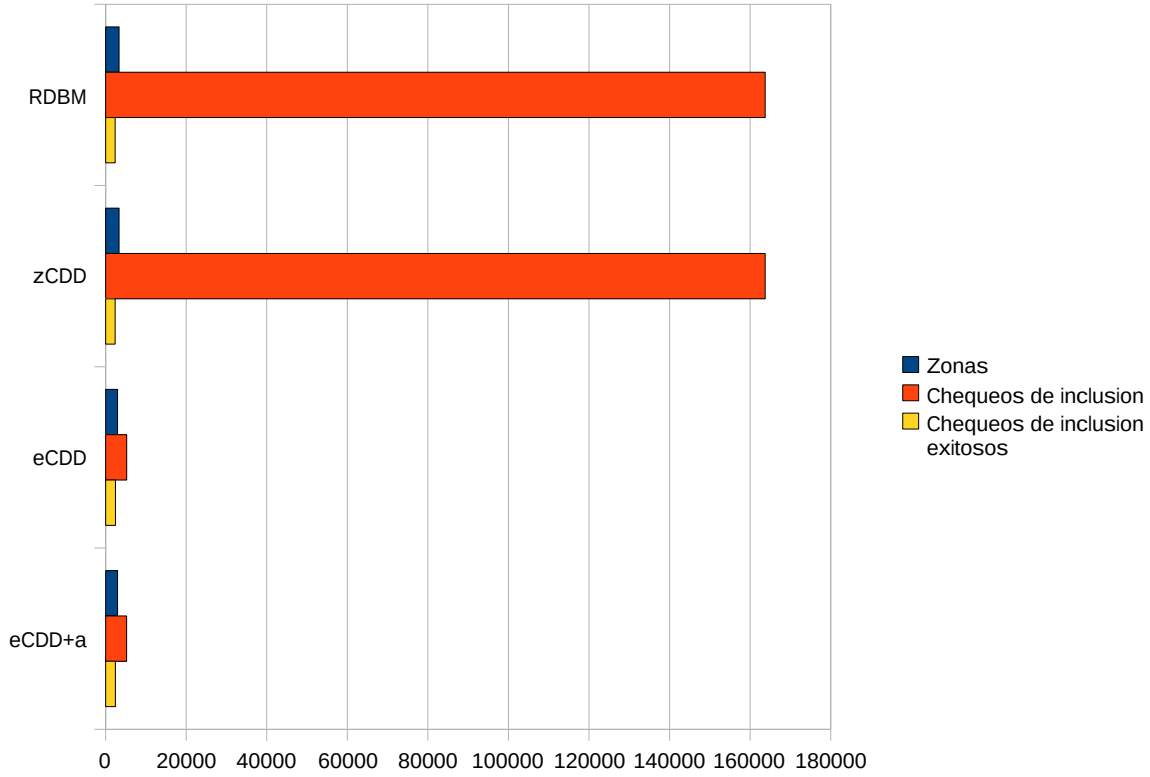


Figura 5.13: RCS4 True: Zonas y Chequeos de inclusión

5.2.4. Struct-ObsSliced

En este caso se observa la misma diferencia abismal entre zCDD/RDBM y eCDD/eCDD +a. También puede notarse que eCDD +a ahorra una gran cantidad de espacio frente a eCDD, pero el tiempo transcurrido se mantiene igual.

5.2.5. Conveyor6-AB

Nuevamente se percibe una gran mejora en tiempo y espacio entre eCDD y eCDD +a.

5.2.6. Consideraciones finales

Acerca de las versiones *False* de los casos de prueba estudiados

Puede observarse que todos los resultados presentados corresponden a casos de prueba en su versión *True*, esto es, en los que es posible alcanzar la propiedad buscada.

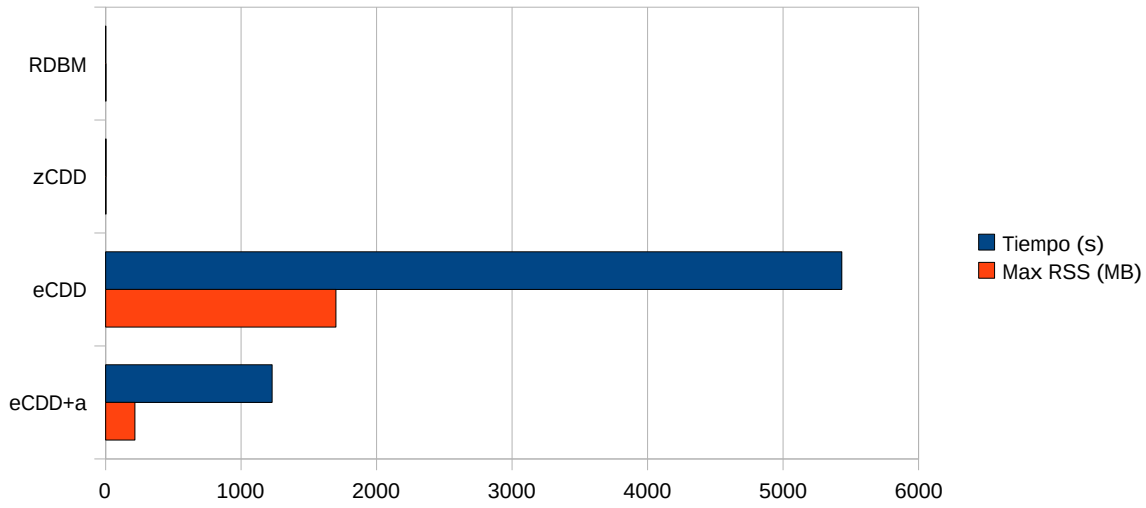


Figura 5.14: RCS4 True: Tiempo y espacio

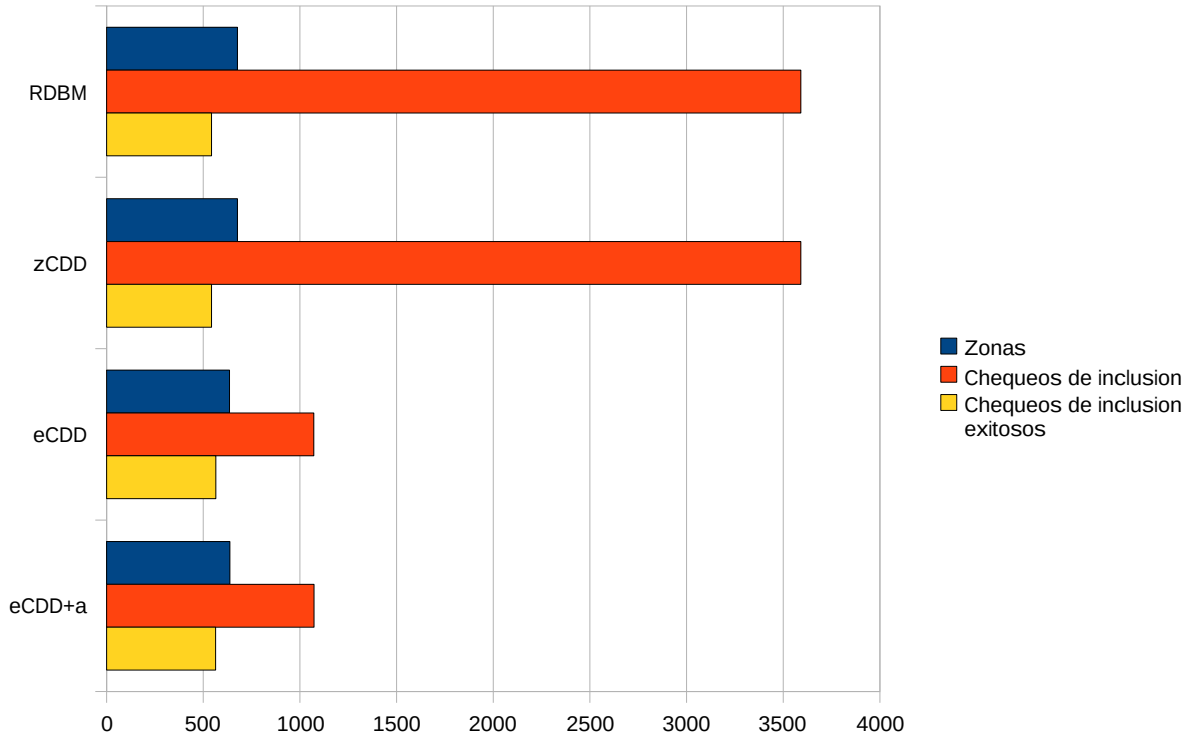


Figura 5.15: Struct ObsSliced True: Zonas y Chequeos de inclusión

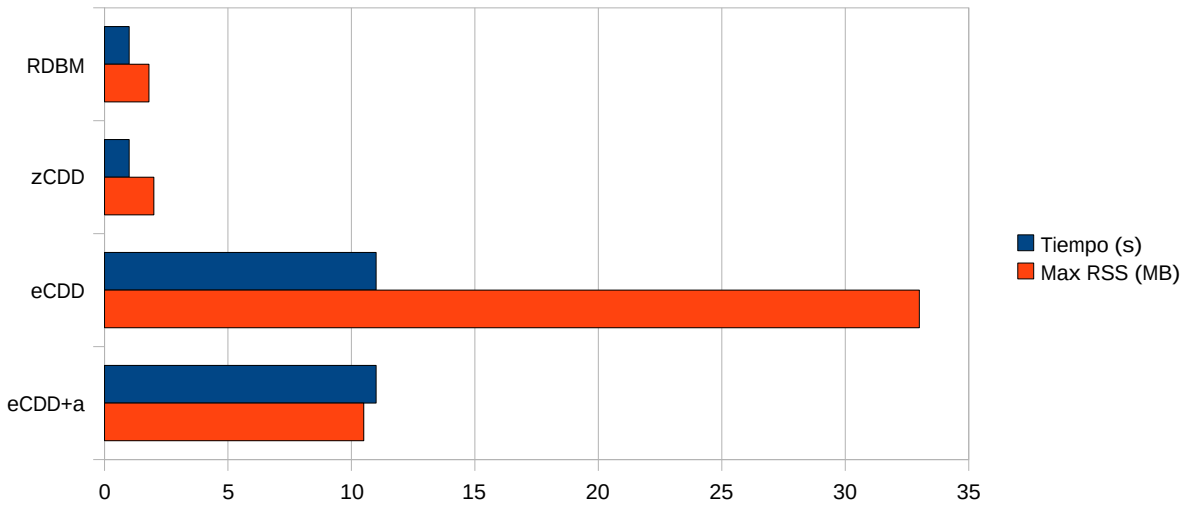


Figura 5.16: Struct ObsSlicedTrue: Tiempo y espacio

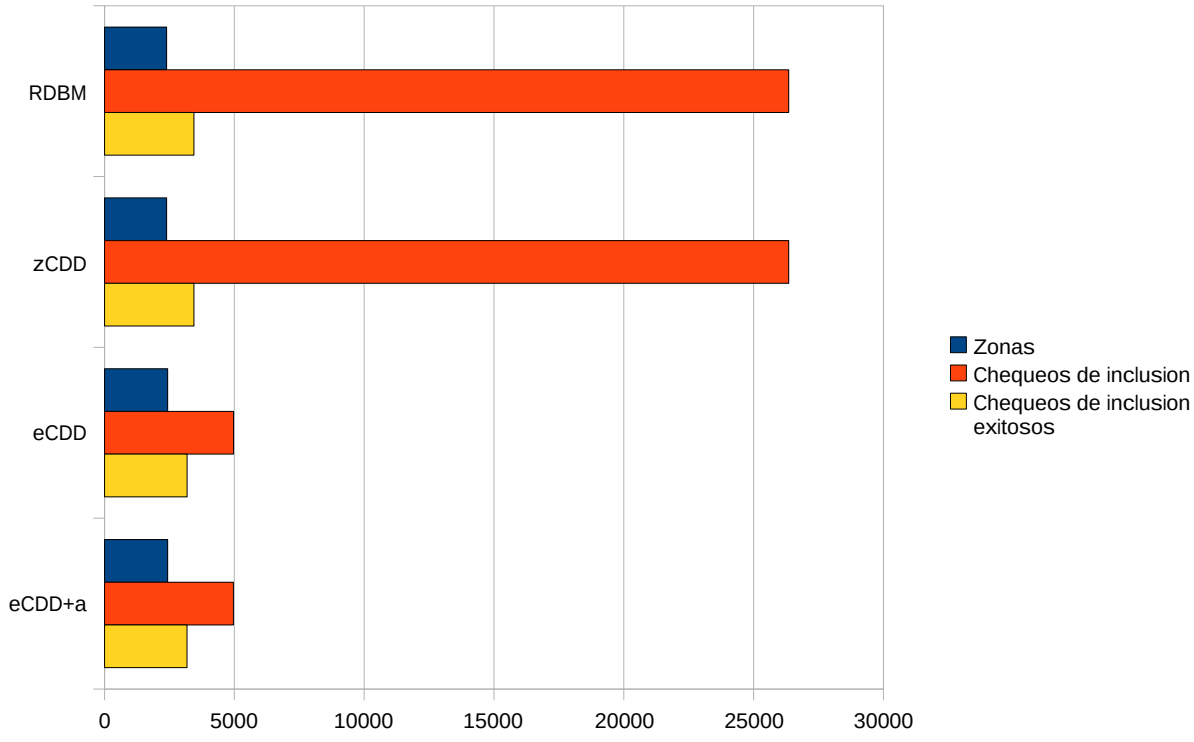


Figura 5.17: Conveyor6-AB True: Zonas y Chequeos de inclusión

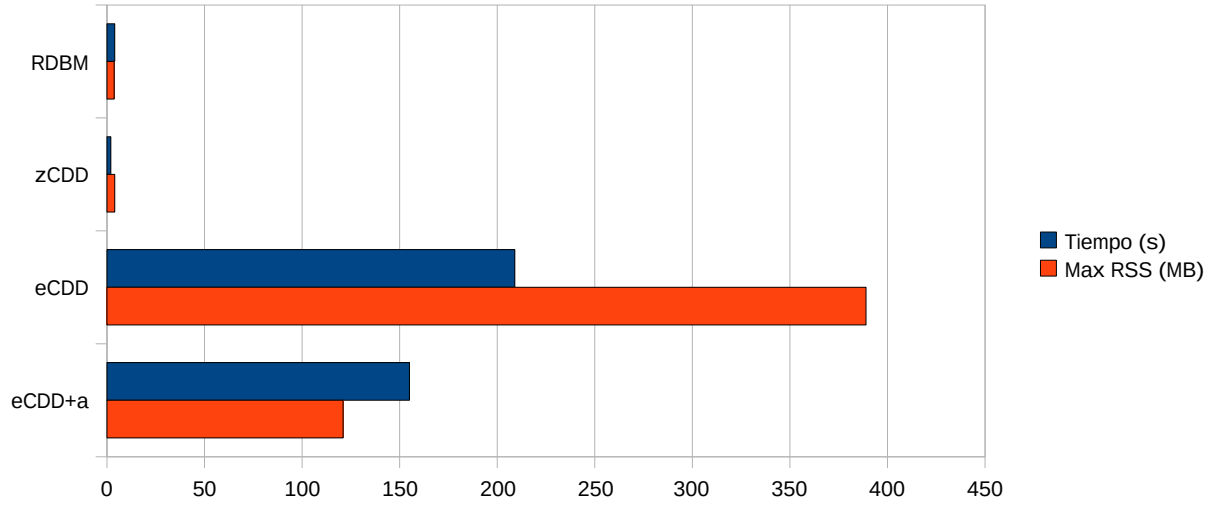


Figura 5.18: Conveyor6-AB True: Tiempo y espacio

En los casos *False* —por el contrario— no se alcanza la propiedad buscada, lo que obliga al verificador a recorrer todo el espacio de búsqueda. Esto deviene en un consumo mayor de tiempo y espacio para completar la verificación.

No reportamos corridas de casos *False* pues para las versiones *e*CDD y *e*CDD +a todos los casos estudiados terminan consumiendo toda la memoria disponible (*out of memory*, OOM) lo cual no nos permite sacar conclusiones con respecto al beneficio obtenido al hacer aliasing (*e*CDD contra *e*CDD +a).

El único caso False de nuestras pruebas que terminó (y lo hizo para *e*CDD +a pero no para *e*CDD) fue Struct ObsSliced False. En la tabla 5.2 puede verse ejemplos de corridas para los casos FDDI4 False (el más pequeño de la serie FDDI), y el mencionado Struct ObsSliced False.

	Caso de estudio			
	FDDI4		Struct ObsSliced	
Verificador	Tiempo(s)	Memoria (MB)	Tiempo(s)	Memoria (MB)
RDBM	166	7	3	2
zCDD	62	8	2	3
*e*CDD	N/A	OOM	N/A	OOM
*e*CDD +a	N/A	OOM	15494	3474

Cuadro 5.2: Comparación de las distintas implementaciones del model checker para FDDI4 y Struct ObsSliced, ambos False.

El caso de *e*CDD para FDDI4 es notable porque consumió la memoria disponible en menos de 30 segundos.

Con respecto a Struct ObsSliced, aunque *e*CDD +a terminó, lo hizo casi al borde del OOM (que para todos los verificadores estaba fijado en 3500MB).

Acerca del comportamiento de RCS4 True y Conveyor6-AB True

En estos dos casos, a diferencia de los demás, puede verse una mejora de tiempo a favor de *e*CDD +a en comparación con *e*CDD, cuando en los demás casos de prueba el tiempo de *e*CDD +a

era levemente peor. Esta diferencia puede explicarse teniendo en cuenta por separado los efectos las dos optimizaciones implementadas:

- La eliminación de subestructuras por medio de aliasing permite ahorrar espacio a costa de un leve *overhead* dado por el tiempo de compresión y de búsqueda en la tabla de eCDDs.
- La fusión de intervalos iguales (ver 3.1.3) efectivamente ahorra la copia de un eCDD cada vez que es aplicada, sin costo extra (por lo tanto ahorra a la vez tiempo y espacio).

Para un caso de prueba en el que mayormente se ahorre espacio por medio del aliasing —habiendo relativamente pocas uniones de intervalos iguales— es esperable ver un ahorro de espacio a costa de un leve incremento de tiempo.

Por contraste, un caso de prueba en el que la optimización de unir intervalos se aplique frecuentemente puede disminuir el tiempo de ejecución, si esta ganancia supera al costo de eliminar redundancia por aliasing.

El caso de RCS4 es un ejemplo bien marcado de ésto, pues con sólo habilitar la unión de intervalos iguales (sin habilitar aliasing) ya se divide por 5 (aproximadamente) el espacio consumido, comparando contra eCDD (sin unión de intervalos iguales ni aliasing). El agregado de aliasing, para este caso de prueba, dio un comparativamente pequeño margen extra de ahorro de espacio.

Para el caso de Conveyor6-ab ocurre algo similar. Queda pendiente descubrir si hay alguna característica particular de estos casos de prueba que esté generando tantos intervalos iguales unibles.

Capítulo 6

Conclusiones

En este capítulo comentaremos los puntos destacables del desarrollo de la presente Tesis.

En primer lugar, se analizó el comportamiento de la estructura eCDD usada para la representación de regiones temporales en el contexto del model checking temporizado. Se midió la redundancia interna de la estructura que, al ser ésta arbórea, consistió en la proporción de subárboles repetidos presentes con respecto al total de subárboles presentes.

Las mediciones de redundancia dejaron al descubierto un problema interno secundario de la estructura; se estaban repitiendo eCDDs idénticos en intervalos contiguos (y por ende, susceptibles de ser unidos). Esto no sólo desperdiciaba tiempo y espacio en copias innecesarias, sino que perturbaba los hashes de los eCDDs padres del eCDD que tuviera este comportamiento. La detección y prevención de este comportamiento patológico ya de por sí significó una importante mejora en el espacio consumido, además de estabilizar los hashes para poder aplicar aliasing.

Detectada una cantidad significativa de redundancia, se diseñó e implementó de manera efectiva un método de reducción de la redundancia presente en el eCDD, utilizando mecanismos de *aliasing* para tal fin. El método implementado no minimiza el tiempo necesario, pero da una cota superior preliminar del porcentaje de mejora que puede esperarse obtener al aplicar mecanismos de aliasing a esta estructura de datos.

La implementación de aliasing para la estructura mejoró notablemente el rendimiento espacial del uso de eCDDs con unión. En casos como *RCS4*, significó la diferencia entre terminar o no la ejecución del *model checker* en computadoras como la usada para el desarrollo de esta Tesis (aunque en el servidor usado para las mediciones presentes en el capítulo de resultados, esto no resultó un problema, ya que contaba con cuatro veces más memoria RAM).

Es importante tener también en cuenta las desventajas que presenta la implementación de aliasing en su estado actual:

- El método actual utiliza una comparación optimista de hashes para decidir la igualdad entre CDDs. La posibilidad de existencia de colisiones en casos más grandes puede invalidar los resultados obtenidos por el model checker, por lo que para ambientes de producción es necesario realizar comparaciones profundas entre eCDDs. Esto es factible mediante el uso de la comparación presentada en (3.1.3), aunque ciertamente encarece el tiempo de ejecución.
- La tabla de hash implementada para recordar los eCDDs generados tiene un tamaño máximo acotado, y tiene limitada la cantidad de eCDDs con el mismo hash (colisiones) que puede almacenar. Esto puede resultar en una detección incompleta de subestructuras duplicadas. En este sentido puede ser necesario realizar ajustes finos a la tabla para priorizar según algún criterio la inserción de eCDDs en la tabla (para "recordar" eCDDs que sean mejores candidatos que otros para *aliasing*).

- El porcentaje de espacio ganado mediante esta técnica, aunque notable, no compensa la diferencia de espacio de órdenes de magnitud observada entre RDBMs/zCDDs y eCDDs con unión.

Ante este escenario, la técnica desarrollada no parece lista para reemplazar a las RDBMs tradicionales o a los eCDDs sin unión. El espacio consumido sigue siendo muy superior al de las ya mencionadas. Suponemos que la redundancia interna no es el único problema a resolver en el uso de eCDDs con unión.

Creemos además que el trabajo realizado es importante ya que permite echar luz sobre un aspecto de las estructuras arbóreas que no ha sido suficientemente analizado en el contexto de los model checkers temporizados: el manejo de aliasing. Si bien trabajos anteriores ofrecen una perspectiva más positiva al respecto [BLP+99] destacamos la validación independiente como uno de los pilares de la ciencia. En este caso se ve nuevamente demostrada su utilidad.

Con respecto a la redundancia, una posible línea de estudio que tenemos en cuenta es que, al ser los hashes puramente sintácticos, nos estemos perdiendo de hacer *aliasing* entre sub-eCDDs que sean *semánticamente* iguales, a pesar de que sus hashes correspondientes sean distintos. Para poner a prueba esta idea puede hacer falta comparar eCDDs mediante un chequeo de doble inclusión u algo similar. Una dificultad de esto es que la detección de candidatos a alias sería más complicada que en el caso del uso de hashes.

Otros trabajos realizados [Wan03] sugieren que el orden entre relojes definido puede influir significativamente en el tamaño final de un eCDD dado, por lo que un posible trabajo futuro sería encontrar métodos para elegir un orden entre relojes óptimo, manteniendo incluso un orden entre relojes diferente para cada locación visitada.

Bibliografía

[ACD+92] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of the 13th IEEE Real-time Systems Symposium*, pages 157–166, Phoenix, Arizona, 1992.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[Alu92] Rajeev Alur. *Techniques for automatic verification of real-time systems.* PhD thesis, Stanford University, Stanford, CA, USA, 1992.

[Bal96] F. Balarin. Approximate reachability analysis of timed automata. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 52, Washington, DC, USA, 1996. IEEE Computer Society.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, volume 1427, pages 546–550. Springer-Verlag, 1998.

[BGO04a] Víctor Braberman, Diego Garbervetsky, and Alfredo Olivero. ObsSlice: A timed automata slicer based on observers. In *Proc. of the 16th Intl. Conf. CAV '04*, LNCS. Springer Verlag, 2004.

[BGO04b] Víctor A. Braberman, Diego Garbervetsky, and Alfredo Olivero. Obsslice: A timed automata slicer based on observers. In *CAV*, pages 470–474, 2004.

[BLL+96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL – a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid Systems III: Verification and Control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[BLP+99] Gerd Behrmann, Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using Clock Difference Diagrams. In *Computer Aided Verification*, pages 341–353, 1999.

[BOS05] Víctor Braberman, Alfredo Olivero, and Fernando Schapachnik. Issues in Distributed Model-Checking of Timed Automata: building ZEUS. *International Journal of Software Tools for Technology Transfer*, 7:4–18, feb 2005.

[BPO03] Victor Braberman, Carlos G. Lopez Pombo, and Alfredo Olivero. On improving backwards verification of timed automata. In Eugene Asarin, Oded Maler, and Sergio Yovine, editors, *Proceedings of Workshop on Theory and Practice of Timed Systems (TPTS-ETAPS) '02*, volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BY] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools.

[Daw98] Conrado Daws. Optikron: A tool suite for enhancing model-checking of real-time systems. In *Computer Aided Verification*, pages 542–545, 1998.

[Dil90] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 197–212, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 208–219, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[EMA+97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, M. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In O. Maler, editor, *Hybrid and Real-Time Systems*, pages 346–360, Grenoble, France, 1997. Springer Verlag, LNCS 1201.

[HHWT95] Thomas A. Henzinger, Pei Ho, and Howard Wong-Toi. A user guide to HyTech. Technical report, Cornell University, Ithaca, NY, USA, 1995.

[KL98] K. Strehl and L. Thiele. Symbolic model checking using Interval Diagram techniques. Technical report, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), 1998.

[Pav06] Esteban Pavese. Una nueva estructura de datos basada en bdds para el model checking temporizado. Master's thesis, Universidad de Buenos Aires, 2006.

[REC+93] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE /ACM International Conference on CAD*, pages 188–191, Santa Clara, California, 1993. IEEE Computer Society Press.

[Sch02] Fernando P. Schapachnik. *Verificación distribuida y paralela de sistemas de tiempo real.* Tesis de licenciatura, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Buenos Aires, Argentina, Mayo 2002.

[Sch07] Fernando Schapachnik. *Timed Automata Model Checking in Monoprocessor and Multiprocessor Architectures.* Phd thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, October 2007.

[Wan97] Thomas Wang. Integer hash functions. http://www.cris.com/~ttwang/tech/inthash.htm, 1997.

[Wan00] Farn Wang. Efficient data structure for fully symbolic verification of real-time software systems. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 157–171, London, UK, 2000. Springer-Verlag.

[Wan03] Farn Wang. Efficient verification of timed automata with BDD-like data-structures. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 189–205, London, UK, 2003. Springer-Verlag.

[Yov96] Sergio Yovine. Model checking timed automata. In *European Educational Forum: School on Embedded Systems*, pages 114–152, 1996.