Trabajo Final Lic. en Ciencias de la Computación "Optimización en Dominios de Planificación"

Año 2013

Estudiante: Lic. Facundo Bustos
Facultad de Matemática, Astronomía y Física (FaMAF)
Universidad Nacional de Córdoba, Argentina (UNC)
Tel. Fijo: (0054) 351 4815465 / Tel. Móvil: 54-9-351-6154632
email: facundojosebustos@gmail.com

Director: Dr. Carlos Areces
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
Facultad de Matemática, Astronomía y Física (FaMAF)
Universidad Nacional de Córdoba, Argentina (UNC)
Tel: (0054) 351 5353701 Int.41414 o 43030
email: carlos.areces@gmail.com

Trabajo Final Lic. en Ciencias de la Computación "Optimización en Dominios de Planificación"

Alumno: Lic.Facundo Bustos
Director: Dr. Carlos Areces
FaMAF – Universidad Nacional de Córdoba

Año 2013

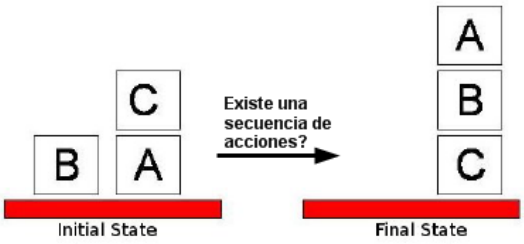
Resumen

El problema de la planificación es un problema de decisión que se estudia dentro del área de Inteligencia Artificial (IA). Consiste básicamente en, dado un conjunto finito de acciones (llamado dominio), un estado inicial y un objetivo, decidir si existe o no una secuencia finita de tales acciones que me permita pasar del estado inicial a un estado en el cual se satisface el objetivo. Actualmente existen algoritmos, llamados planificadores, que realizan esta tarea, como por ejemplo, Fast Foward (FF) [1].

Nuestro trabajo consistirá en encontrar una transformación de un problema de planificación PP en otro problema de planificación PP', donde los tiempos de respuesta del planificador sean menores en PP' que en PP y además teniendo que existir la propiedad de que el conjunto de soluciones entre ambos problemas de planificación se preservan, es decir, que un problema tenga solución si y sólo si el otro también y que podamos recuperar una solución de PP a través de una solución de PP' y viceversa. Vamos a explicar en detalle la técnica de transformación llamada Split y luego presentamos una formalización de la misma que nos permite demostrar formalmente propiedades deseables de nuestra técnica. Proponemos cuatro algoritmos que automatizan la técnica, analizando sus ventajas y desventajas. Analizamos y discutimos los resultados obtenidos (usando FF como planificador) en diez dominios de prueba, comparando la performance del dominio Split con respecto a la del dominio original. Finalmente, presentamos tres posibles aplicaciones de nuestra técnica de optimización.

1. El problema de la planificación

Un problema de planificación consiste en decidir si existe o no una secuencia finita de acciones que me permita llegar de un estado inicial a un estado en el cual se satisface un determinado objetivo.



La planificación es un ejercicio de control de la explosión combinatoria, si tenemos un número p de literales en un dominio tendremos 2^p estados. Para dominios complejos, p puede crecer mucho. *La planificación es una tarea compleja, de hecho, se conoce que pertenece al conjunto de problemas de complejidad PSpace-completo, lo cual, nos da una noción de que es común o muy frecuente que un planificador no pueda terminar de decidir si existe o no tal secuencia de acciones. Esto último deja en evidencia la importancia de hallar una técnica que permita una optimización en el tiempo de respuesta del planificador.*

1.1. Conceptos Básicos de un Problema de Planificación

Estados. Conjunto de literales positivos. Por ejemplo, {*En(avion*$_1$*, ezeiza), En(avion*$_2$*, aeroparque)*} puede representar un estado en un problema de reparto de encomiendas, en donde el $avion_1$ se encuentra en el aeropuerto de Ezeiza mientras que el $avion_2$ se encuentra en el aeropuerto de Aeroparque. La hipótesis de un mundo cerrado es asumida por lo que todas las condiciones que no son mencionadas en un estado se asumen que son falsas.

Objetivos. Conjunto de literales positivos, donde no se asume la hipótesis de un mundo cerrado, es decir, no sabemos nada acerca del valor de verdad de los literales que no ocurren en el objetivo.

Acciones. Permiten pasar de un estado a otro y se especifican en términos de las precondiciones que deben cumplirse para poder ser ejecutada y de los efectos que producen cuando se ejecutan. Ejemplo:

$AccionVolar(P : Avion, Desde, Hacia : Aeropuerto)$
$pre : \{En(P, Desde)$
$add : \{En(P, Hasta)\}$
$del : \{En(P, Desde)\}$

Diremos que una acción es ejecutable en un estado si este satisface su precondición y el resultado de ejecutar una acción en un estado s nos lleva a un estado w, que es igual a s excepto que cualquier literal positivo en add es agregado a w y cualquier literal positivo en del es eliminado de w. El ejemplo anterior no es una acción propiamente dicha sino un esquema de acción, es decir, un esquema de acción expresa varias acciones posibles, una por cada instanciación posible de sus variables.

1.2. Formalizando el Problema de Planificación

Necesitamos formalizar el problema de planificación par luego probar propiedades deseables de nuestra técnica.

Una acción a es una 3-upla (pre, add, del) donde pre, add, del conjuntos (finitos) de literales positivos instanciados tales que $add \cap del = \emptyset$ (notar que pensamos las acciones como ya instanciados, no como esquemas). Definimos el conjunto de literales como *Lit*(a)= $pre(a) \cup add(a) \cup del(a)$ y el espacio de estados definido por un dominio Σ_A = $\mathcal{P}$(*Lit*(A)).

Definición 1. *Un problema de planificación PP será una 3-upla (A, s_0, F) donde*
$A = \{a_1, ..., a_n\}$ *un conjunto finito no vacío de acciones, llamado dominio.*
$s_0 \in \Sigma_A$, *llamado el estado inicial.*
$F \subseteq \Sigma_A$, *llamado el conjunto de estados finales.*

Definimos la siguiente relación de transición, sea $a \in A$ y $s, w \in \Sigma_A$, $s \xrightarrow{a} w$ sii $pre(a) \subseteq s$ y $w = (s \cup add(a)) \setminus del(a)$. Y su clausura transitiva, sea $\overline{a} = a_1...a_n$ definimos, $s \xrightarrow{\overline{a}} w$ sii $\exists s_0...s_n \in \Sigma_A$. $s_0 \xrightarrow{a_1} s_1...s_{n-1} \xrightarrow{a_n} s_n$ con $s_0 = s$ y $s_n = w$.

Definimos el conjunto de soluciones de un problema de planificación,
Sol(*PP*) = $\{\overline{a} \in A^* : s_0 \xrightarrow{\overline{a}} s_f$, con $s_f \in F\}$. Finalmente, si $\overline{a} \in$ *Sol*(*PP*) sii $\overline{a}$ es un plan de PP.

2. Noción de "equivalencia" entre Dominios de Planificación

Nuestra intención es definir una noción de equivalencia entre dominios tal que podamos obtener un plan de uno de ellos a través de un plan del otro y viceversa. Una primer definición de equivalencia podría ser aquella tal que si dos dominios A y A' son equivalentes entonces todo plan de A es también plan de A' y viceversa. Una definición así tiene el defecto de que las acciones de ambos dominios deberían ser las mismas, es decir, obliga a que el espacio de soluciones de problemas de planificación que tienen a A y A' como dominios sean el mismo. La idea es poder definir una noción de equivalencia entre dominios que pueden ser completamente diferentes y que además sea lo más general posible. En otras palabras, *lo que nos interesa es que los dominios posean un espacio de soluciones equivalente*, es decir, siempre que en uno de los dominios haya solución esto también ocurra en el otro y viceversa. Además para poder utilizar uno de los dominios como alternativa del otro, vamos a pedir que sea posible traducir una solución de uno de ellos en una solución del otro. Como consecuencia, necesitamos introducir el concepto de traducción de un plan de A en un plan de A' y viceversa. También necesitamos una traducción de los estados que define A hacia los estados que define A'. La idea de nuestra definición es que, si uno pasa de un estado s a un estado w mediante una secuencia de acciones $\overline{a}$ de A, entonces debe existir una traducción de $\overline{a}$ en una secuencia de acciones $\overline{a'}$ de A' tal que permita pasar del estado que es traducción de s a un estado que es traducción de w. Recíprocamente, si tenemos una secuencia de acciones $\overline{a'}$ de A' que me lleva de un estado que es traducción de un estado s de A a un estado que es también traducción de un estado w de A, entonces debe existir una traducción de $\overline{a'}$ en una secuencia de acciones $\overline{a}$ de A que me lleve de s a w. Es importante destacar que esta condición no se necesita que se cumpla para cualesquiera dos estados s y w, sino para ciertos estados que son los que se van a configurar finalmente como estado inicial y final para un problema de planificación. Por lo tanto restringiremos la

noción de equivalencia a un conjunto de pares de posibles estados iniciales y finales. A continuación, presentamos nuestra definición de equivalencia entre dominios de planificación.

Definición 2. *Sean A, A' dominios, $C \subseteq \Sigma_A \times \Sigma_A$ un conjunto de pares de estados, $T : \Sigma_A \to \Sigma_{A'}$, $t : A^* \to A'^*$, $t' : A'^* \to A^*$. Diremos que A y A' son equivalentes vía T, t, t' en el contexto C (notación, $A \equiv_C^{T,t,t} A'$) sii*

I. $\forall (s, w) \in C \,.\, \forall \overline{a} \in A^* \,.\, s \xrightarrow{\overline{a}} w \Rightarrow T(s) \xrightarrow{t(\overline{a})} T(w)$

II. $\forall (s, w) \in C \,.\, \forall \overline{a'} \in A'^* \,.\, T(s) \xrightarrow{\overline{a'}} T(w) \Rightarrow s \xrightarrow{t'(\overline{a'})} w.$

Notar que la relación $\equiv_C^{T,t,t}$ no es una relación de equivalencia. Si bien es reflexiva (basta con tomar T, t, t' como la identidad) no es necesariamente simétrica ni transitiva. Para obtener estas últimas dos propiedades deberíamos, por ejemplo, tomar C como el conjunto de todos pares de estados y requerir que T sea biyectiva (para asegurar que T^{-1} esté bien definida). Pero estos requisitos no son necesarios para asegurar equivalencia del espacio de soluciones que es lo que buscamos (ver Teorema 1) y además hacen menos general la definición.

El teorema que sigue (demostración 1 en el apéndice B) asegura que problemas de planificación con dominios equivalentes preservan el conjunto de soluciones. Es decir, garantiza que podemos recuperar un plan de uno de los problemas de planificación a través de la traducción de un plan del otro.

Teorema 1. *Si $A \equiv_C^{T,t,t} A'$, entonces tomando $PP = (A, s_0, F)$ y $PP' = (A', T(s_0), T(F))$ problemas de planificación tales que $\forall s_f \in F : (s_0, s_f) \in C$, se cumple que:*

I. $t(Sol(PP)) \subseteq Sol(PP')$

II. $t'(Sol(PP') \subseteq Sol(PP)$.

3. Split de un dominio de planificación

En esta sección, presentamos una técnica para obtener dominios equivalentes y además queremos que el planificador en el nuevo dominio funcione "mejor" que en el dominio original, logrando así la optimización que buscábamos. Qué vamos a entender por "mejor"? Primero vamos a aclarar, que cuando corremos un problema de planificación en un planificador éste último puede terminar (diciendo si hay plan o no) ó no puede terminar. Entonces, lo mejor sería que si el planificador no termina con el dominio original entonces ahora si termine con el nuevo dominio ó si el planificador termina con el dominio original también lo haga con el nuevo dominio pero ahora más rápidamente.

3.1. Split a través de un ejemplo

La técnica, denominada Split, consiste básicamente en tomar cada acción del dominio original y dividirla en partes o subacciones tal que al ejecutar las subacciones en forma atómica se tenga los mismos efectos que ejecutar la acción original. Veamos como se le aplica Split a la siguiente acción:

$Move(A, B, C : Bloque)$
$pre : \{on(A, B), clear(A), clear(C)\}$
$add : \{on(A, C), clear(B)\}$
$del : \{on(A, B), clear(C)\}$

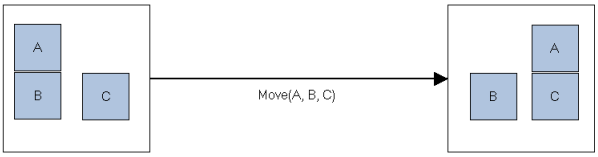


Figura 1: Ejecución de la acción $Move(A, B, C)$.

Y la dividimos en dos subacciones:

$Move_1(A, B : Bloque)$	$Move_2(A, C : Bloque)$
$pre : \{on(A, B), clear(A)\}$	$pre : \{clear(C)\}$
$add : \{clear(B)\}$	$add : \{on(A, C)\}$
$del : \{on(A, B)\}$	$del : \{clear(C)\}$

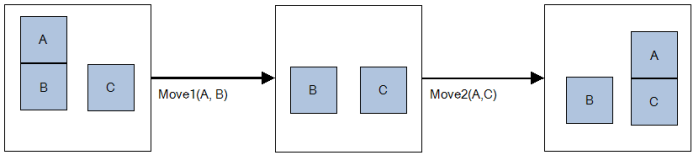


Figura 2: Ejecución de la acción $Move_1(A, B)$ seguido de $Move_2(A, C)$.

Por un lado, observar que ejecutar la acción $Move_1$ seguido de $Move_2$ es equivalente a ejecutar la acción $Move$. Pero que ganamos haciendo esta división? Si el tipo $Bloque$ tiene k instancias entonces la acción $Move$ de tres variables en su interfaz genera k^3 instanciaciones del esquema de acción mientras que $Move_1$ seguido de $Move_2$, ambas con dos variables en sus interfaces, genera $k^2 + k^2$ instanciaciones. Por ejemplo, para un valor de $k = 100$ tenemos 1.000.000 instanciaciones, mientras que usando las subacciones tenemos 20.000. *Este ahorro en las instanciaciones de los esquemas de acción que debe realizar el planificador es lo que esperamos que optimice los tiempo del respuesta del mismo.*

Por otro lado, al dividir las acciones en subacciones hacemos que el dominio de las subacciones (el dominio Split) pueda ser más expresivo que el dominio original, es decir, el dominio Split puede generar la aparición de más planes como consecuencia de esta división y recordar que nosotros siempre queremos que los conjuntos de soluciones se mantengan equivalentes. Para esto debemos eliminar la posibilidad de que aparezcan planes que no puedan codificarse en una solución del dominio original, para ello utilizaremos literales nuevos como tokens que se irán pasando entre las subacciones. Veamos como quedan las subacciones $Move_1$ y $Move_2$ colocados los literales tokens:

$Move_1(A, B : Bloque)$	$Move_2(A, C : Bloque)$
$pre : \{on(A, B), clear(A), procnone\}$	$pre : \{clear(C), do^{Move_2}, arg(A)\}$
$add : \{clear(B), do^{Move_2}, arg(A)\}$	$add : \{on(A, C), procnone\}$
$del : \{on(A, B), procnone\}$	$del : \{clear(C), do^{Move_2}, arg(A)\}$

El token $procnone$ desactiva la ejecutabilidad de subacciones que pertenezcan a otras acciones, no queremos que ocurra interleaving entre subacciones de diferentes acciones, es decir, necesitamos que las subacciones de un Split se ejecuten en forma atómica. El token do sirve para otorgar los turnos para que se ejecuten las subacciones de una misma acción. Por último, el token Arg sirve para asegurar que si una subacción instancia una variable que es variable de una subacción posterior en el Split, estas dos subacciones deben instanciar dicha variable con la misma instancia.

3.2. ¿Cómo obtenemos un Split a partir de un esquema de acción?

Queremos automatizar la técnica de Split. Un algoritmo de Split realiza dos tareas. Primero debe obtener un cubrimiento de la interfaz del esquema de acción original. Cada uno de los elementos del cubrimiento serán la interfaces de las subacciones (por lo tanto, la cardinalidad del cubrimiento me determina la cantidad de subacciones del Split). La segunda tarea es tomar un mapeo entre los literales del esquema de la acción original y los elementos del cubrimiento. Definamos el concepto de cubrimiento.

Definición 3. *Sea a una acción, $C \subseteq \mathcal{P}(interfaz(a))$ no vacío es un cubrimiento de a sii*

I. $\bigcup_{c \in C} c = interfaz(a)$

II. $\forall c, c' \in C.\ c \subseteq c' \Rightarrow c = c'$.

III. $\forall l \in Lit(a) : \exists c \in C : interfaz(l) \subseteq c$.

Podemos pensar un cubrimiento como una partición de la interfaz de la acción original donde se permite overlapping de las particiones. En particular, en el ejemplo de la acción $Move$ de la sección 3.1 cuya interfaz es $\{A, B, C\}$, tomamos el cubrimiento $C_0 = \{\{A, B\}, \{A, C\}\}$. Notar que C_0 satisface la definición anterior.

Resta determinar qué literales de la acción original tendrá cada una de las subacciones, es para ello que definimos un mapeo entre los literales de la acción original y las subacciones cuyas interfaces están ya definidas por los elementos del cubrimiento. Mapearemos la ocurrencia de un literal l a la subacción a_i con interfaz $c \in C$ si l es cubierto por c (i.e la interfaz del literal l está incluida en c). Notar que dado un literal este puede ser cubierto por más de un elemento de C. Obviamente, el mapeo respeta el lugar de ocurrencia del literal dentro de la acción original. Es decir, si un literal ocurre en la sección pre, add ó del de la acción original dicho literal ocurrirá en la sección pre, add ó del respectivamente, de la subacción a la que fue mapeado. Es muy importante que la definición del mapeo cumpla con la siguiente condición: si un literal l ocurre en la precondición y en el efecto de la acción original entonces ambas ocurrencias deben mapearse al mismo elemento de C. Esta condición tiene como objetivo evitar concretamente las siguientes dos situaciones:

- que una subacción elimine un literal que está también en la precondición de otra subacción que se va a ejecutar posteriormente convirtiendo a esta última en una subacción no ejecutable independientemente de la ejecutabilidad de la acción original.

- que una subacción agregue un literal que está también en la precondición de otra subacción que se va a ejecutar posteriormente convirtiendo a esta última en una subacción ejecutable independientemente de la ejecutabilidad de la acción original.

Veamos un ejemplo de la primer situación:

$XX(A, B, C : Bloque)$
$pre : \{on(A, B), clear(A), clear(C)\}$
$add : \{on(C, A)\}$
$del : \{clear(A)\}$

y la dividimos en dos:

$XX_1(A, B : Bloque)$	$XX_2(A, C : Bloque)$
$pre : \{on(A, B)\}$	$pre : \{clear(A), clear(C)\}$
$add :$	$add : \{on(C, A)\}$
$del : \{clear(A)\}$	$del :$

Supongamos que la acción XX es ejecutable en PP en un cierto estado, por lo tanto XX_1 seguido de XX_2 debería ser ejecutables en PP' pero luego de ejecutar XX_1, XX_2 no es ejecutable, pues $clear(A)$ está en su pre y este literal fue removido por XX_1. Claramente esto se debe a que las dos ocurrencias de $clear(A)$ de la acción original se mapearon a diferentes subacciones.

3.3. La importancia del orden de ejecución de las subacciones del Split

Observemos que dada una acción a y $a_1, ..., a_k$ un Split de a, cualquier permutación $a_{i_1}, ..., a_{i_k}$ es un Split de a. Nos preguntamos, habrá en general una permutación más conveniente que otra?

Supongamos que la acción a no es ejecutable en algún estado. Entonces sabemos que existe un literal en su precondición que es falso, y ese mismo literal está en la precondición de una única subacción a_j. Entonces esta subacción también será no ejecutable en algún estado. Entonces si hacemos que a_j sea la última de las subacciones del Split ($j = i_k$), el planificador instanciaría el resto de las $k - 1$ subacciones para darse cuenta de que esta última subacción no se puede ejecutar. En cambio si aj es la primera subacción ($j = i_1$) del Split, el planificador ahorraría las $k - 1$ instanciaciones de las demás subacciones. Es decir, que lo más conveniente sería que la subacción que no se puede ejecutar esté lo antes posible en el orden de la permutación. En general, no sabemos de antemano cual es la subacción que será no ejecutable en caso de que la acción sea no ejecutable. Como heurística podemos considerar que mientras más literales haya en la precondición de una subacción más probabilidad tiene de ser no ejecutable. Por lo tanto, ordenaremos el Split de forma tal que las subacciones con más literales en su precondición estén al principio de la permutación.

3.4. Obteniendo la transformación de un problema de planificación

Hemos visto hasta ahora como dado un problema de planificación PP obtenemos un Split (con tokens) de cada una de las acciones en el dominio de PP. Nos resta especificar cómo se construye el problema de planificación PP' a partir de PP el cual usaremos para recuperar una solución de PP mediante una solución de PP'.

- Los literales en PP' serán los literales de PP más los literales nuevos que se usaron como tokens.
- El dominio de PP' estará formado por las subacciones que conforman los Splits de cada una de las acciones del dominio de PP.
- El estado inicial de PP' será el estado inicial de PP más el literal token $procnone$.
- El objetivo de PP' será el objetivo de PP más el literal token $procnone$.
- Los tipos y sus instancias en PP' serán los mismos que en PP (i.e la transformación preserva los tipos y sus instancias).

3.5. Recuperando un solución de PP a través de su transformación PP' y viceversa

Si $a_1...a_n$ es una solución de PP y $a_i^1, ..., a_i^{k_i}$ es el Split con tokens de la acción a_i, entonces $a_1^1, ..., a_1^{k_1}, ... , a_n^1, ... , a_n^{k_n}$ será una solución en PP'.

Si tenemos una solución de PP, tiene que ser de la forma $a_1^1, ..., a_1^{k_1}, ... , a_n^1, ... , a_n^{k_n}$ donde $a_i^1, ..., a_i^{k_i}$ es el Split con tokens de la acción a_i del dominio de PP, entonces $a_1, ..., a_n$ es una solución de PP.

3.6. Split y equivalencia, parte formal

El objetivo de esta sección es demostrar que un dominio y su Split satisfacen nuestra noción de equivalencia. Definimos formalmente el concepto de Split.

Definición 4. *Sea A un dominio. Una función Split sobre A será una función $\alpha : A \to Acciones^*$, tal que si $\alpha(a) = a_1...a_k$ se cumple que:* [1]

I. $\forall l \in \text{Lit}(A).\ l \in X(a) \Leftrightarrow \exists!\ a_i :\ l \in X(a_i)$ *con* $X \in \{pre, add, del\}$

II. $\forall l \in \text{Lit}(A).\ l \in pre(a) \cap X(a) \Rightarrow \exists!\ a_i :\ l \in pre(a_i) \cap X(a_i)$ *con* $X \in \{add, del\}$

La primera condición dice que los literales de la acción original se mapean a una única subacción y todo literal de una subacción es literal de la acción original. Mientras que la segunda condición dice que si en una acción original un literal ocurre en la pre y en el efecto, ya sea en add ó del, entonces ambas ocurrencias se mapean a la misma subacción. De la definición de Split se desprenden las siguientes proposiciones.

Proposición 1. *Sea $\alpha(a) = a_1...a_k$ una función Split y $X \in \{pre,\ add,\ del\}$.*

1. $X(a_i) \subseteq X(a)$.
2. $X(a_i) \cap X(a_j) = \emptyset$, *con* $i \neq j$.
3. $\bigcup X(a_i) = X(a)$.
4. $pre(a_i) \cap add(a_j) = \emptyset$, *con* $i \neq\ j$.

[1] donde $Acciones$ es el conjunto de todas las acciones.

5. $pre(a_i) \cap del(a_j) = \emptyset$, *con* $i \neq j$.

6. $add(a_i) \cap del(a_j) = \emptyset$.

Definimos formalmente la función para implementar el sistema de tokens:
$\Gamma : A^* \rightarrow A'^*$
$\Gamma(a_1...a_k) = a'_1...a'_k$ donde

caso $k = 1$:
$a'_1 = (pre(a_1) \cup \{procnone\},\ add(a_1),\ del(a_1))$
caso $k > 1$:
$a'_1 = (pre(a_1) \cup \{procnone\},\ add(a_1) \cup \{do^{a'_2}\},\ del(a_1) \cup \{procnone\})$
$a'_j = (pre(a_j) \cup \{do^{a'_j}\},\ add(a_j) \cup \{do^{a'_{j+1}}\},\ del(a_j) \cup \{do^{a'_j}\})$ *con* $1 < j < k$
$a'_k = (pre(a_k) \cup \{do^{a'_k}\},\ add(a_k) \cup \{procnone\},\ del(a_k) \cup \{do^{a'_k}\})$
con $\{do^{a'_i}\}, \{procnone\} \notin Lit(A)$.

El token $arg()$ no ocurre en la definición de la función Γ porque, recordemos, que en la formalización del problema de planificación pensamos en acciones instanciadas, no en esquemas de acción. A continuación definimos formalmente el concepto de dominio Split.

Definición 5. *Sea A un dominio y* α *una función Split sobre A.* $Split_\alpha(A) = \{a' : \exists\ a \in A\ tq\ a' \in \tilde{\alpha}(a)\}$, *con* $\tilde{\alpha} = \Gamma \circ \alpha$. [2]

El siguiente teorema es uno de los resultados principales de este trabajo: dice que un dominio y su Split son equivalentes vía ciertas traducciones en cualquier contexto (demostración 2 en Apéndice B).

Teorema 2. *Sea A un dominio y* α *una función Split sobre A, entonces* $A \equiv_C^{T,t,t'} Split_\alpha(A)$ *con* $C \subseteq \Sigma_A \times \Sigma_A$, $T(s) = s \cup \{procnone\}$, $t(a_1...a_n) = \tilde{\alpha}(a_1)...\tilde{\alpha}(a_n)$ *y* t' *tal que* $t'(t(\overline{a})) = \overline{a}$.

4. Algoritmos de Split

En esta sección se presentan diferentes algoritmos que realizan Split de una acción. Recordar que un algoritmo de Split consiste básicamente en dos tareas: la primera es hallar un cubrimiento de la interfaz de la acción original (cuyos elementos serán las interfaces de las subacciones) y la segunda definir un mapeo entre los literales de la acción original y los elementos de dicho cubrimiento.

4.1. Algoritmo de cubrimiento: Literal Fit

Este algoritmo obtiene un cubrimiento cuyos elementos son lo más chicos posibles para obtener subacciones con interfaces "angostas" y de esta forma generar menor cantidad de instanciaciones. Para esto, vamos a partir del conjunto vacío ($\emptyset$) e ir agregando elementos a medida que los literales de la acción original lo demanden. Es decir, los elementos del cubrimiento se van ajustando a las interfaces de los literales de la acción original. A continuación presentamos el algoritmo:

[2] Notación: $[\overline{a}]_i$ es el i-ésimo elemento de la secuencia de acciones $\overline{a}$ y $a \in \overline{a}$ *sii* $\exists i : [\overline{a}]_i = a$ (sobrecargamos el operador de pertenencia).

Entrada: a una acción
Salida: C un cubrimiento de $interfaz(a)$

```
1: C ← ∅
2: for all l ∈ Lit(a) do
3:     if no existe c ∈ C tq interfaz(l) ⊆ c then
4:         C ← C ∪ {interfaz(l)}
5:         for all c ∈ C do
6:             if c ⊂ interfaz(l) then
7:                 C ← C \ {c}
8:             end if
9:         end for
10:    end if
11: end for
12: return C
```

Observar que recorremos los literales de la acción original chequeando si la interfaz del literal ya es cubierto por un elemento del cubrimiento, en cuyo caso continuamos con el siguiente literal, en caso contrario agregamos a la interfaz del literal como un nuevo elemento del cubrimiento y luego eliminamos los elementos del cubrimiento que ahora son cubiertos estrictamente por este nuevo elemento, así sucesivamente hasta recorrer todos literales de la acción.

4.2. Algoritmo de cubrimiento: Optimized Literal Fit

La principal desventaja de Literal Fit es que al intentar minimizar las interfaces de las subacciones tenemos el costo de generar muchas subacciones. A continuación definimos una función de costo que no sólo pondera el tamaño de las interfaces de las subacciones sino que también pondera la cantidad de subacciones que produce un cubrimiento:

$$Costo(C) = |C| + max_{c \in C}|c| \tag{1}$$

Observar que el cubrimiento obtenido con Literal Fit tiende a minimizar el segundo término de la ecuación 1 y a maximizar el primero. También observar que si hacemos merge de dos elementos de un cubrimiento y eliminamos los elementos que ahora son subconjuntos del merge sigue siendo un cubrimiento. Formalmente, sea C un cubrimiento y $c, c' \in C$, entonces $(C \setminus \{d \in C : d \subseteq c \cup c'\}) \cup \{c \cup c'\}$ es un cubrimiento.

Ahora vamos a querer un cubrimiento que minimice la suma de ambos términos de la ecuación 1. Para ello partimos del cubrimiento Literal Fit y en cada iteración obtenemos un nuevo cubrimiento, que se obtiene de hacer merge de dos elementos del cubrimiento anterior que más reducen la función de costo sin perder la minimización del segundo término de la ecuación 1 lograda con Literal Fit. Así sucesivamente hasta que no se pueda reducir más la función de costo. A continuación, presentamos el algoritmo:

Entrada: a una acción
Salida: C un cubrimiento de $interfaz(a)$

```
1: C' ← LiteralFit(a)
2: gr ← gr(C')
```

3: **repeat**
4: $C \leftarrow C'$
5: $suc \leftarrow Sucesores(C')$
6: **for all** $C'' \in suc$ **do**
7: **if** $Costo(C'') < Costo(C') \wedge gr(C'') = gr$ **then**
8: $C' \leftarrow C''$
9: **end if**
10: **end for**
11: **until** $C' = C$
12: **return** C

Observar que en cada iteración obtenemos (si hay) un sucesor que disminuye el costo pero sin perder la optimización del segundo término de la ecuación 1 obtenida en LiteralFit.

4.3. Algoritmo de cubrimiento: Π-Merger

Vamos a abandonar la estrategia greedy introduciendo una heurística para decidir qué dos elementos de un cubrimiento vamos a mergear en cada iteración. La idea es generar una sucesión de cubrimientos $C_1, ..., C_t$ donde C_1 es el cubrimiento calculado en Literal Fit, C_{i+1} se obtiene haciendo merge de dos elementos de C_i y C_t es el cubrimiento singleton que representa la acción sin dividirse. Finalmente, el cubrimiento devuelto por el algoritmo será el C_i cuyo costo asociado sea el menor. ¿Cuál es la heurística para decidir cuáles dos elementos harán merge? La idea es tomar el merge que tenga más probabilidades de no aumentar el segundo término de la función de costo. Para ello definimos la siguiente función que pretende medir cuánto overlapping hay entre dos elementos de un cubrimiento. Si tenemos mucho overlapping entonces fusionar esos dos elementos no genera una interfaz muy "ancha". Definimos la heurística que tomaremos:

$$\Pi(c_1, c_2) = \frac{|c_1 \cap c_2|}{|c_1 \cup c_2|} \tag{2}$$

Notar que $0 \leq \Pi(c_1, c_2) \leq 1$. Si $\Pi(c_1, c_2) \sim 0$ significa que la intersección de c_1 y c_2 es muy pequeña en comparación con su unión, por lo tanto, existe una mayor probabilidad de que el merge de ambos elementos aumente el segundo término de la ecuación 1. Mientras que si $\Pi(c_1, c_2) \sim 1$ significa que la intersección es bastante parecida a su unión, por lo tanto, el merge entre ambos elementos tiene menor probabilidad de aumentar el segundo término de la ecuación 1. Calcularemos la función Π para cada par de elementos del cubrimiento, generando una matriz simétrica (ya que, $\Pi(c_1, c_2) = \Pi(c_2, c_1)$) y haremos merge de los dos elementos que tengan valor máximo en la función Π, ignorando los elementos de la diagonal. Presentamos el algoritmo:

Entrada: a una acción
Salida: C un cubrimiento de $interfaz(a)$
1: $C' \leftarrow LiteralFit(a)$
2: $C \leftarrow C'$
3: **while** $|C'| > 1$ **do**
4: $Calcular\ mat(C')$
5: $Obtener\ algún\ (i, j)\ tal\ que\ mat(C')[i, j]\ es\ máximo\ con\ i \neq j$

6: $c' = c_i \cup c_j$
7: $C'' \leftarrow (C' \setminus \{d \in C' : d \subseteq c'\}) \cup \{c'\}$
8: **if** $Costo(C'') \leq Costo(C)$ **then**
9: $C \leftarrow C''$
10: **end if**
11: $C' \leftarrow C''$
12: **end while**
13: **return** C

4.4. Algoritmo de mapeo

Una vez obtenido el cubrimiento resta definir el mapeo entre los literales de la acción original y los elementos del cubrimiento. En este caso tomaremos la siguiente política de mapeo: un literal de la acción original se mapea al elemento del cubrimiento de mayor tamaño que lo cubre. A continuación el algoritmo:

Entrada: C un cubrimiento de la acción a
Salida: map un mapeo entre literales de a y elementos de C
1: $C \leftarrow sort(C)$
2: **for all** $l \in Lit(a)$ **do**
3: **for all** $c \in C$ **do**
4: **if** $interfaz(l) \subseteq c$ **then**
5: $map(l) \leftarrow c$
6: **break**
7: **end if**
8: **end for**
9: **end for**
10: **return** map

Cabe aclarar que $sort(C)$ ordena los elementos de C de mayor a menor con respecto a la cardinalidad de los mismos.

5. Pruebas y análisis de los resultados obtenidos con los algoritmos de Split

En esta sección vamos a analizar los resultados obtenidos de los diferentes algoritmos de Split que ya presentamos. Dado un algoritmo de Split, vamos a comparar cómo funciona éste con respecto al dominio original para una muestra de diez dominios de prueba (bajados de la web de ICAPS [2]). Para esto vamos a definir una métrica que compara la performance del dominio original y su Split indicando cuál de los dos dominios funcionó mejor. El planificador que escogimos para realizar estas pruebas es Fast Foward (FF) [1]. Recordemos que el planificador en un problema de planificación puede terminar dando una solución (hay plan o no hay plan) ó no terminar (timeout). Definimos la métrica: sean t, t' el tiempo de respuesta del planificador cuando encuentra un plan en el dominio original D y en el dominio Split D' respectivamente.

$$\Delta(t,t') = \begin{cases} \log(\frac{t+err}{t'+err}) & si\ hay\ plan\ tanto\ en\ D\ como\ en\ D' \\ 5 & si\ timeout\ en\ D\ y\ hay\ solución\ en\ D' \\ -5 & si\ hay\ solución\ en\ D\ y\ timeout\ en\ D' \\ 0 & si\ (timeout\ en\ D\ y\ D')\ ó\ (no\ hay\ plan\ en\ D\ y\ D') \end{cases}$$

De la definición de Δ se observa que:

- si Δ= 0 significa que tanto el dominio original como el Split se comportaron iguales, no hubo diferencias.
- si Δ= 5 significa el mejor caso posible, ya que, el dominio original no pudo terminar (timeout) mientras que el dominio Split terminó dando una solución (ya sea, hay plan o no hay plan).
- si Δ= -5 significa el peor caso posible, ya que, el dominio original terminó dando una solución (ya sea, hay plan o no hay plan) mientras que el Split no pudo terminar (timeout).
- si $0 < \Delta < 5$ significa un caso positivo, ya que, ambos dominios terminaron dando un plan, pero el dominio Split lo hizo más rápidamente que el dominio original. Notar que, mientras Δ más se acerca a 5 ($\Delta \to 5$) mejor es la optimización lograda con el Split.
- si $-5 < \Delta < 0$ significa un caso negativo, ya que, ambos dominios terminaron dando un plan pero el dominio original lo hizo más rápidamente que el dominio Split. Notar que, mientras Δ más se acerca a -5 ($\Delta \to -5$) peor es el comportamiento del Split con respecto al dominio original.
- La constante err es un término de error que representa el error de medición. Sirve para evitar la división por cero porque es siempre mayor a cero y lo suficientemente chica como para ser despreciable con respecto a las magnitudes de t y t'.

Las gráficas que vamos a analizar, constan en el eje horizontal con los problemas (par estado inicial y objetivo) de los diez dominios, mientras que en el eje vertical tenemos el valor Δ para esos problemas. Vamos a tener una curva por cada uno de los diez dominios de prueba.

En la figura 3 del apéndice A se muestra la gráfica Delta para el algoritmo Literal Fit. En los dominios d3, d6 Literal Fit converge al dominio original, por lo tanto, no se muestran en la gráfica. Esto último sugiere uque podemos utilizar a Literal Fit como herramienta correctora de modularidad (ver sección 6). En la gráfica se observa que la mayor densidad se encuentra sobre el eje horizontal -5, lo que significa que tenemos gran cantidad de casos donde el dominio original terminó con una solución y Literal Fit no. Existe densidad considerable debajo del eje 0, lo que significa que ambos dominios terminaron con un plan pero funcionó mejor el dominio original. Se observa cierta densidad sobre el eje 0, lo que significa que tenemos casos donde el dominio original y Literal Fit se comportaron iguales. Existe muy poca densidad arriba del eje 0, lo que significa que no hubo prácticamente casos donde Literal Fit funcionó mejor que el dominio original. Dos problemas (el problema 5 para d4 y el problema 42 para d10) donde Split nos permitió encontrar solución a problemas que antes desconocíamos su solución. En el problema 13 para d10 y en el 41 para d7 Split nos permite encontrar una solución en menor tiempo.

En la figura 4 del apéndice A se muestra la gráfica Delta para el algoritmo Optimized Literal Fit. En los dominios d3, d6 Optimized Literal Fit converge al dominio original, por lo tanto, no se muestran en la gráfica. En la gráfica se observa claramente un incremento de puntos por encima del eje 0 para el caso del dominio d10 con respecto a la gráfica de Literal Fit. Esto es importante

porque podemos encontrar soluciones que ya se tenían con el dominio original pero ahora en forma más veloz. En el dominio d7, aparecen 4 puntos por encima del eje 0 y esto es importante porque logramos tener en 4 problemas la solución más rápidamente. También aparecen 13 problemas del dominio d7 donde el dominio original no terminaba (timeout) y Optimized termina encontrando una solución. Esto último es muy valioso, ya que, había al menos 13 problemas en los cuales se desconocían sus soluciones y Optimized permite que si se conozcan. Estos son los casos que sugieren utilizar Optimized en forma concurrente (ver sección 6). Por otro lado, continúa existiendo una densidad importante debajo del eje 0 y sobre el eje -5, lo que indica que en varios problemas el dominio original funciona mejor que Optimized Literal Fit.

En la figura 5 del apéndice A se muestra la gráfica Delta para el algoritmo Π-Merger. En los dominios d2, d3, d4, d6, d8, d9 Π-Merger converge al dominio original, por lo tanto, no se muestran en la gráfica. En la gráfica observamos que prácticamente en todo el conjunto de problemas del dominio d1, Π-Merger funciona mucho mejorar que el dominio original. Del problema 1 al 38 la mejora oscila alrededor de un orden de magnitud (recordar que la escala es logarítmica base 10), mientras que de los problemas 39 al 60 la mejora está entre uno a tres órdenes de magnitud. Estos son los casos que sugieren utilizar Π-Merger en forma off-line (ver sección 6). Existen cuatro problemas (22, 26, 28, 36) del dominio d10 donde Π-Merger permite conocer la solución mientras que el dominio original no. Existe densidad sobre el eje -5 pero es visiblemente menor al caso de las gráficas anteriores (Literal Fit y Optimized). Esto quiere decir que cuando Π-Merger no llega a optimizar los tiempos de respuesta tampoco provoca que el planificador no pueda terminar. No se observa resultado positivo en ninguno de los problemas para el dominio d5.

6. Aplicaciones de Split

Split en modo concurrente. Si tenemos la posibilidad de contar con dos microprocesadores y en uno de ellos corremos el planificador con el dominio original y en el otro corremos el planificador con el dominio Split, entonces, logramos que en los problemas del dominio original donde el planificador no terminaba (timeout) y por lo tanto no conocíamos su solución ahora tengo chances de encontrarla con el dominio Split. Y en los problemas del dominio original donde el planificador terminaba y encontraba una solución ahora tengo chances de que el dominio Split también lo haga pero más rápidamente. *Es decir, conviene invertir en agregar un segundo microprocesador que en mejorar el que ya teníamos*. Más general aún, podemos tener un microprocesador por cada algoritmo de Split que tengamos (el hardware es barato). De esta forma siempre estaremos mejorando los tiempos de respuesta del planificador. Además Split en modo concurrente permite dejar de interpretar en forma negativa los puntos que se encontraban debajo del eje 0 en las tres gráficas del apéndice A, ahora nos resultan indiferentes, en el sentido de que en esos problemas estamos igual que como estábamos antes de implementar la técnica de optimización, es decir, esos problemas son resueltos por el dominio original mientras que los otros problemas son resueltos por el dominio Split en una forma más óptima. Podemos decir que en algunos problemas estamos como en el principio y en varios otros hemos mejorado, por lo tanto, en total hemos mejorado.

Split en modo off-line. Si para un conjunto de problemas (set de problemas de prueba) observamos una mejora generalizada en los tiempos de respuesta del planificador en el dominio Split con respecto al dominio original, podemos inferir que cualquier otro problema tiene una alta probabilidad de que ocurra lo mismo, por lo tanto, conviene correr el planificador en el dominio Split y

olvidarnos del dominio original. Mientras que si no se observa una mejora generaliza planifico con el dominio original. *Split en modo off-line nos permite evitar la concurrencia.* En general: dado un dominio calculamos diferentes Splits y luego veo cómo funcionan éstos en el conjunto de problemas (set de problemas). Si se observa que hay importante proporción de éstos problemas donde se produce una optimización en los tiempos de respuesta para un cierto algoritmo de Split, entonces, planificamos en ese dominio Split y si no planificamos con el dominio original.

Split como herramienta para corregir la falta de modularidad de los dominios. En algunos dominios se observó que los diferentes algoritmos de Split convergieron al dominio original, más precisamente, en aquellos dominios que presentaban acciones con interfaces pequeñas. Mientras que en los dominios con acciones cuyas interfaces eran más grandes Split dividió. De esta forma, *podemos pensar a Split como una herramienta que devuelve un dominio más modularizado si es que detecta la falta de ello*.

7. Conclusiones

A continuación listamos fortalezas y debilidades de nuestra técnica de optimización.

- Fortalezas de Split:
  - Utilización en modo offline.
  - Utilización en modo concurrente.
  - Utilización como herramienta para corregir la falta de modularidad de los dominios, preservando la expresividad de los mismos, ya que, en los dominios con interfaces pequeñas Split tiende a converger al dominio original.
  - Realiza trade-off entre branch y profundidad en el grafo de planificación, Split reduce el branch pero incrementa la profundidad, lo cual puede resultar ventajoso en ciertos dominios.
- Debilidades de Split:
  - No asegura en general un dominio mejor que el original, por ejemplo, de los diez dominios de prueba sólo uno de ellos arrojó una mejora general contundente (d1 con Π-Merger).
  - No asegura reducir la cantidad de instanciaciones totales realizadas finalmente por el planificador.
  - Puede producir una optimización no homogénea (en ciertos problemas de un mismo dominio optimiza y en otros no).

Referencias

[1] Inteligencia Artificial, un enfoque moderno, 2da edición, capítulo 11. Stuart J. Russell y Peter Norvig. Editorial Pearson Prentice Hall.

[2] Web ICAPS: `http://ipc.icaps-conference.org/`

A. Apéndice: Gráficas Métrica Δ

Las gráficas de este apéndice, constan en el eje horizontal de los problemas (par estado inicial y objetivo) de los diez dominios, mientras que en el eje vertical tenemos el valor Δ para esos problemas. Vamos a tener una curva por cada uno de los diez dominios de prueba (si para un dominio particular el Split converge al dominio original dicha curva no se muestra en la gráfica.

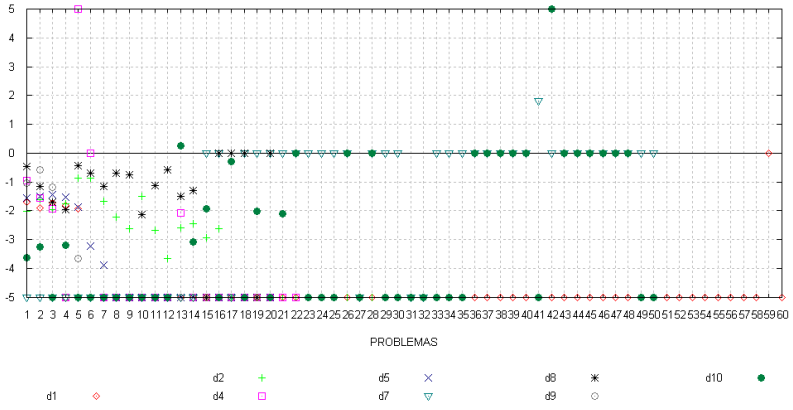


Figura 3: Gráfica Delta para Literal Fit.

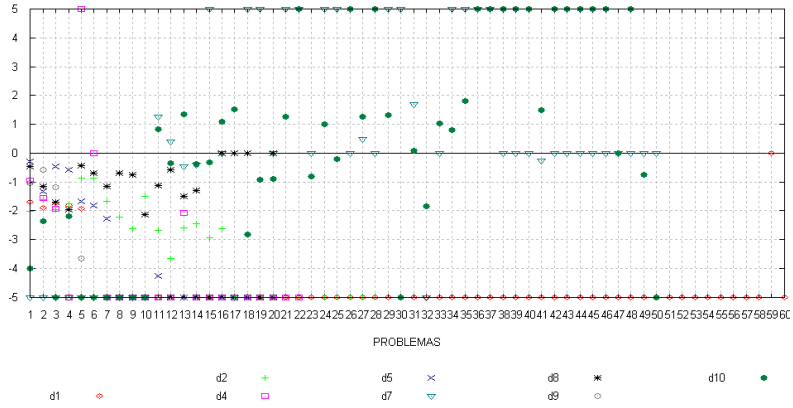


Figura 4: Gráfica Delta para Optimized Literal Fit.

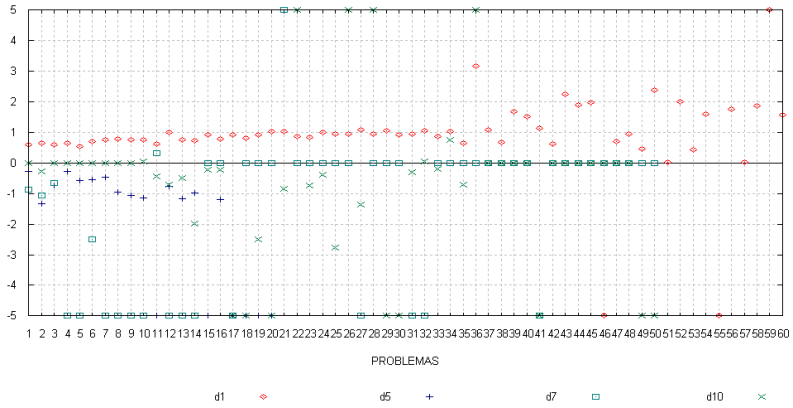


Figura 5: Gráfica Delta para Π-Merger.

B. Apéndice: Demostraciones Formales

Demo 1. *(del Teorema 1 sección 2)*

Veamos que si $\overline{a} \in Sol(PP)$*, entonces* $t(\overline{a}) \in Sol(PP')$.

Sea $\overline{a} \in Sol(PP)$

$\Rightarrow s_0 \xrightarrow{\overline{a}} s_f$ *con* $s_f \in F$

$\Rightarrow T(s_0) \xrightarrow{t(\overline{a})} T(s_f)$ *por def. de equivalencia*

$\Rightarrow t(\overline{a}) \in Sol(PP')$

Veamos que si $\overline{a'} \in Sol(PP')$*, entonces* $t'(\overline{a'}) \in Sol(PP)$.

Sea $\overline{a'} \in Sol(PP')$

$\Rightarrow T(s_0) \xrightarrow{\overline{a'}} T(s_f)$ *para algún* $s_f \in F$

$\Rightarrow s_0 \xrightarrow{t'(\overline{a'})} s_f$ *por def. de equivalencia*

$\Rightarrow t'(\overline{a'}) \in Sol(PP)$

Demo 2. *(del Teorema 2 sección 3.6)*

Primero vemos si se cumple que:

$\forall (s_0, s_f) \in C \,.\, \forall \overline{a} \in A^* \,.\, s_0 \xrightarrow{\overline{a}} s_f \Rightarrow T(s_0) \xrightarrow{t(\overline{a})} T(s_f)$.

Sea $(s_0, s_f) \in C$ *y* $\overline{a} \in A^*$ *tal que* $s_0 \xrightarrow{\overline{a}} s_f$

$\Rightarrow s_0 \xrightarrow{a_1} s_1 ... s_{n-1} \xrightarrow{a_n} s_f$

$\Rightarrow T(s_0) \xrightarrow{t(a_1)} T(s_1) ... T(s_{n-1}) \xrightarrow{t(a_n)} T(s_f)$ *por Lema 3*

$\Rightarrow T(s_0) \xrightarrow{t(a_1)...t(a_n)} T(s_f)$

$\Rightarrow T(s_0) \xrightarrow{t(a_1...a_n)} T(s_f)$

$\Rightarrow T(s_0) \xrightarrow{t(\overline{a})} T(s_f)$

Por último, vemos si se cumple que:

$\forall (s_0, s_f) \in C \,.\, \forall \overline{a'} \in Split_\alpha(A)^* \,.\, T(s_0) \xrightarrow{\overline{a'}} T(s_f) \Rightarrow s_0 \xrightarrow{t'(\overline{a'})} s_f$.

Sea $(s_0, s_f) \in C$ *y* $\overline{a'} \in Split_\alpha(A)^*$ *tal que* $T(s_0) \xrightarrow{\overline{a'}} T(s_f)$

$\Rightarrow T(s_0) \xrightarrow{t(\overline{a})} T(s_f)$ *para algún* $\overline{a} \in A^*$ *por Lema 4*

$\Rightarrow T(s_0) \xrightarrow{t(a_1...a_n)} T(s_f)$

$\Rightarrow T(s_0) \xrightarrow{t(a_1)...t(a_n)} T(s_f)$

$\Rightarrow T(s_0) \xrightarrow{t(a_1)} s'_1 ... s'_{n-1} \xrightarrow{t(a_n)} T(s_f)$

$\Rightarrow T(s_0) \xrightarrow{t(a_1)} T(s_1) ... T(s_{n-1}) \xrightarrow{t(a_n)} T(s_f)$ *por Lema 2*

$\Rightarrow s_0 \xrightarrow{a_1} s_1 ... s_{n-1} \xrightarrow{a_n} s_f$ *por Lema 3*

$\Rightarrow s_0 \xrightarrow{a_1...a_n} s_f$

$\Rightarrow s_0 \xrightarrow{\overline{a}} s_f$

$\Rightarrow s_0 \xrightarrow{t't(\overline{a})} s_f$ *def. de t'*

$\Rightarrow s_0 \xrightarrow{t'(\overline{a'})} s_f$

Lema 1. *Sea A un dominio, $a \in A$, $s \in \Sigma_A$, $\alpha(a) = a_1...a_k$ función Split sobre A y T, $t(a) = a'_1...a'_k$ las funciones de traducción definidas en el Teorema 2.* [3]

1. *Si a es ejecutable en s, entonces $\forall q < k$, $T(s) \xrightarrow{a'_1...a'_q} w'$ con $w' = (s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}$.*

2. *Si $T(s) \xrightarrow{a'_1...a'_r} s'_r$ para $r < k$, entonces $\forall q \leq r$, $pre_{i \leq q}(a_i) \subseteq s$ y $s'_q = (s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}$.*

Prueba 1. *Veamos la prueba de 1. Hacemos inducción en q.*

Caso q=1: debo ver que $T(s) \xrightarrow{a'_1} w'$ y $w' = ((s \cup add(a_1)) \setminus del(a_1)) \cup \{do^{a'_2}\}$.

$pre(a_1) \subseteq pre(a) \subseteq s$
$\Rightarrow pre(a_1) \cup \{procnone\} \subseteq s \cup \{procnone\}$
$\Rightarrow pre(a'_1) \subseteq T(s)$
$\Rightarrow a'_1$ *ejecutable en* $T(s)$
$\Rightarrow T(s) \xrightarrow{a'_1} w'$
$w' = (T(s) \cup add(a'_1)) \setminus del(a'_1)$
$w' = ((s \cup \{procnone\}) \cup (add(a_1) \cup \{do^{a'_2}\})) \setminus (del(a_1) \cup \{procnone\})$
$w' = ((s \cup add(a_1)) \setminus del(a_1)) \cup \{do^{a'_2}\}$

Caso inductivo: debo ver que si $T(s) \xrightarrow{a'_1...a'_q} v'$ y $v' = (s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}$ entonces $T(s) \xrightarrow{a'_1...a'_{q+1}} w'$ y $w' = (s \cup_{i \leq q+1} add(a_i) \setminus_{i \leq q+1} del(a_i)) \cup \{do^{a'_{q+2}}\}$.

$pre(a_{q+1}) \subseteq pre(a) \subseteq s$
$\Rightarrow pre(a_{q+1}) \subseteq s \cup_{i \leq q} add(a_i)$
$\Rightarrow pre(a_{q+1}) \setminus_{i \leq q} del(a_i) \subseteq s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)$ *por proposición 1*
$\Rightarrow pre(a_{q+1}) \subseteq s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)$ *por proposición 1*
$\Rightarrow pre(a_{q+1}) \cup \{do^{a'_{q+1}}\} \subseteq (s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}$
$\Rightarrow pre(a'_{q+1}) \subseteq v'$
$\Rightarrow a'_{q+1}$ *es ejecutable en* v'
$\Rightarrow T(s) \xrightarrow{a'_1...a'_q} v' \xrightarrow{a'_{q+1}} w'$ y $w' = (v' \cup add(a'_{q+1})) \setminus del(a'_{q+1})$
$\Rightarrow T(s) \xrightarrow{a'_1...a'_{q+1}} w'$ y $w' = (v' \cup add(a'_{q+1})) \setminus del(a'_{q+1})$
$w' = (((s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}) \cup add(a'_{q+1})) \setminus del(a'_{q+1})$
$w' = (((s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}) \cup (add(a_{q+1}) \cup \{do^{a'_{q+2}}\}))$
$\setminus (del(a_{q+1}) \cup \{do^{a'_{q+1}}\})$
$w' = (s \cup_{i \leq q+1} add(a_i) \setminus_{i \leq q+1} del(a_i)) \cup \{do^{a'_{q+2}}\}$ *por proposición 1*

Veamos la prueba de 2. Hacemos inducción en q.

Caso q=1: debo ver que $T(s) \xrightarrow{a'_1...a'_r} s'_r$ para $r < k \Rightarrow pre(a_1) \subseteq s$ y $s'_1 = ((s \cup add(a_1)) \setminus del(a_1)) \cup \{do^{a'_2}\}$.

[3] Sea S, A_i, B_i conjuntos, con $i = 1, ..., n$, denotaremos al conjunto $(S \cup A_1 \cup ... \cup A_n) \setminus B_1 \setminus ... \setminus B_n$ con la expresión $S \cup_{i \leq n} A_i \setminus_{i \leq n} B_i$. Por último, decimos que $\overline{b}$ es tramo inicial propio de $\overline{a}$ *sii* $\overline{a} = \overline{b}\,\overline{a'}$ y $\overline{b} \neq \varepsilon, \overline{a}$.

$T(s) \xrightarrow{a'_1 ... a'_r} s'_r$
$T(s) \xrightarrow{a'_1} s'_1$
$pre(a'_1) \subseteq T(s)$ *y* $s'_1 = (T(s) \cup add(a'_1)) \setminus del(a'_1)$
$pre(a_1) \cup \{procnone\} \subseteq s \cup \{procnone\}$ *y* $s'_1 = ((s \cup \{procnone\}) \cup add(a'_1)) \setminus del(a'_1)$
$pre(a_1) \subseteq s$ *y* $s'_1 = ((s \cup \{procnone\}) \cup (add(a_1) \cup \{do^{a'_2}\})) \setminus (del(a_1) \cup \{procnone\})$
$pre(a_1) \subseteq s$ *y* $s'_1 = ((s \cup add(a_1)) \setminus del(a_1)) \cup \{do^{a'_2}\}$.

Caso inductivo: debo ver que $T(s) \xrightarrow{a'_1 ... a'_r} s'_r$ *para* $r < k \Rightarrow pre_{i \leq q+1}(a_i) \subseteq s$ *y* $s'_{q+1} = (s \cup_{i \leq q+1} add(a_i) \setminus del_{i \leq q+1}(a_i)) \cup \{do^{a'_{q+2}}\}$.

$T(s) \xrightarrow{a'_1 ... a'_r} s'_r$
$T(s) \xrightarrow{a'_1 ... a'_{q+1}} s'_{q+1}$ *con* $q+1 \leq r < k$
$T(s) \xrightarrow{a'_1 ... a'_q} s'_q \xrightarrow{a'_{q+1}} s'_{q+1}$
$pre_{i \leq q}(a_i) \subseteq s$ *y* $s'_q = (s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}$ *por hip. inductiva*
$pre(a'_{q+1}) \subseteq s'_q$
$pre(a_{q+1}) \cup \{do^{a'_{q+1}}\} \subseteq (s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}$
$pre(a_{q+1}) \subseteq s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)$
$pre(a_{q+1}) \subseteq s \cup_{i \leq q} add(a_i)$ *por Proposición 1*
$pre(a_{q+1}) \subseteq s$ *por Proposición 1.*

Tenemos que $pre_{i \leq q}(a_i) \subseteq s$ *y* $pre(a_{q+1}) \subseteq s$, *por lo tanto,* $pre_{i \leq q+1}(a_i) \subseteq s$.

$s'_{q+1} = (s'_q \cup add(a'_{q+1})) \setminus del(a'_{q+1})$
$s'_{q+1} = (((s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}) \cup add(a'_{q+1})) \setminus del(a'_{q+1})$
$s'_{q+1} = (((s \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}) \cup (add(a_{q+1}) \cup \{do^{a'_{q+2}}\}))$
$\setminus (del(a_{q+1}) \cup \{do^{a'_q+1}\})$
$s'_{q+1} = (s \cup_{i \leq q+1} add(a_i) \setminus_{i \leq q+1} del(a_i)) \cup \{do^{a'_{q+2}}\}$.

Lema 2. *Sea* A *un dominio,* $a \in A$, $s \in \Sigma_A$, α *una función Split sobre* A *y* T, t *las funciones de traducción definidas en el Teorema 2.*

Si $T(s) \xrightarrow{t(a)} w'$, *entonces* $w' = T(w)$ *para algún* $w \in \Sigma_A$.

Prueba 2. *Tenemos que* $t(a) = \tilde{\alpha}(a) = \Gamma \circ \alpha(a) = \Gamma(a_1 ... a_k) = a'_1 ... a'_k$.

Caso $k = 1$*:*
$T(s) \xrightarrow{a'_1} w'$
$w' = (T(s) \cup add(a'_1)) \setminus del(a'_1)$
$w' = ((s \cup \{procnoce\}) \cup add(a'_1)) \setminus del(a'_1)$
$w' = ((s \cup \{procnoce\}) \cup add(a_1)) \setminus del(a_1)$ *por definición de función* Γ
$w' = ((s \cup add(a)) \setminus del(a)) \cup \{procnoce\}$
$w' = T((s \cup add(a)) \setminus del(a))$ *y* $(s \cup add(a)) \setminus del(a) \in \Sigma_A$.

Caso $k > 1$*:*

$T(s) \xrightarrow{a'_1,\ldots,a'_k} w'$
$T(s) \xrightarrow{a'_1 \ldots a'_{k-1}} s'_{k-1} \xrightarrow{a'_k} w'$ *y* $w' = (s'_{k-1} \cup add(a'_k)) \setminus del(a'_k)$
$s'_{k-1} = (s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)) \cup \{do^{a'_k}\}$ *por Lema 1*
$w' = (s'_{k-1} \cup add(a'_k)) \setminus del(a'_k)$
$w' = (((s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)) \cup \{do^{a'_k}\}) \cup add(a'_k)) \setminus del(a'_k)$
$w' = (((s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)) \cup \{do^{a'_k}\}) \cup (add(a_k) \cup \{procnone\}))$
$\setminus (del(a_k) \cup \{do^{a'_k}\})$
$w' = (s \cup_{i \leq k} add(a_i) \setminus_{i \leq k} del(a_i)) \cup \{procnone\}$ *por Proposición 1*
$w' = ((s \cup add(a)) \setminus del(a)) \cup \{procnone\}$
$w' = T((s \cup add(a)) \setminus del(a))$ *y* $(s \cup add(a)) \setminus del(a) \in \Sigma_A$.

Corolario 1. *Si* $T(s) \xrightarrow{t(\overline{a})} w'$ *entonces* $w' = T(w)$ *para algún* $w \in \Sigma_A$.

Lema 3. *Sea* A *un dominio,* $a \in A$, $s, w \in \Sigma_A$, α *una función Split sobre* A *y* T, t *las funciones de traducción definidas en el Teorema 2.*

$s \xrightarrow{a} w$ *sii* $T(s) \xrightarrow{t(a)} T(w)$.

Prueba 3. *Tenemos que* $t(a) = \tilde{\alpha}(a) = \Gamma \circ \alpha(a) = \Gamma(a_1 \ldots a_k) = a'_1 \ldots a'_k$.

Veamos que $s \xrightarrow{a} w \Rightarrow T(s) \xrightarrow{t(a)} T(w)$.
$s \xrightarrow{a} w$
$pre(a) \subseteq s$ *y* $w = (s \cup add(a)) \setminus del(a)$

Caso $k = 1$*: debo ver que* $T(s) \xrightarrow{a'_1} T(w)$.
$pre(a_1) \subseteq pre(a) \subseteq s$
$\Rightarrow pre(a_1) \cup \{procnone\} \subseteq s \cup \{procnone\}$
$\Rightarrow pre(a'_1) \subseteq T(s)$
$\Rightarrow a'_1$ *ejecutable en* $T(s)$
$\Rightarrow T(s) \xrightarrow{a'_1} w'$ *y* $w' = (T(s) \cup add(a'_1)) \setminus del(a'_1)$
$w' = ((s \cup \{procnone\}) \cup add(a_1)) \setminus del(a_1)$
$w' = ((s \cup \{procnone\}) \cup add(a)) \setminus del(a)$ *por definición de* Γ
$w' = ((s \cup add(a)) \setminus del(a)) \cup \{procnone\}$
$w' = w \cup \{procnone\}$
$w' = T(w)$

Caso $k > 1$*: debo ver que* $T(s) \xrightarrow{a'_1,\ldots,a'_k} T(w)$.

Como a es ejecutable en s tenemos que
$T(s) \xrightarrow{a'_1 \dots a'_{k-1}} s'$ *y* $s' = (s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)) \cup \{do^{a'_k}\}$ *por Lema 1*

Por otro lado,
$pre(a_k) \subseteq pre(a) \subseteq s$
$\Rightarrow pre(a_k) \subseteq s \cup_{i \leq k-1} add(a_i)$
$\Rightarrow pre(a_k) \setminus_{i \leq k-1} del(a_i) \subseteq s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)$
$\Rightarrow pre(a_k) \subseteq s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)$ *por Proposición 1*
$\Rightarrow pre(a_k) \cup \{do^{a'_k}\} \subseteq (s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)) \cup \{do^{a'_k}\}$
$\Rightarrow pre(a'_k) \subseteq s'$
$\Rightarrow a'_k$ *es ejecutable en* s'
$\Rightarrow T(s) \xrightarrow{a'_1 \dots a'_{k-1}} s' \xrightarrow{a'_k} v'$ *y* $v' = (s' \cup add(a'_k)) \setminus del(a'_k)$
$\Rightarrow T(s) \xrightarrow{a'_1 \dots a'_k} v'$ *y* $v' = (s' \cup add(a'_k)) \setminus del(a'_k)$
$v' = (((s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)) \cup \{do^{a'_k}\}) \cup add(a'_k)) \setminus del(a'_k)$
$v' = (((s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)) \cup \{do^{a'_k}\}) \cup (add(a_k) \cup \{procnone\}))$
$\setminus (del(a_k) \cup \{do^{a'_k}\})$
$v' = (s \cup_{i \leq k} add(a_i) \setminus_{i \leq k} del(a_i)) \cup \{procnone\}$ *por Proposición 1*
$v' = ((s \cup add(a)) \setminus del(a)) \cup \{procnone\}$
$v' = w \cup \{procnone\}$
$v' = T(w)$

Veamos que $T(s) \xrightarrow{t(a)} T(w) \Rightarrow s \xrightarrow{a} w$.

Caso $k = 1$*: debo ver que* $T(s) \xrightarrow{a'_1} T(w) \Rightarrow s \xrightarrow{a} w$.
$T(s) \xrightarrow{a'_1} T(w)$
$\Rightarrow pre(a'_1) \subseteq T(s)$ *y* $T(w) = (T(s) \cup add(a'_1)) \setminus del(a'_1)$
$\Rightarrow pre(a_1) \cup \{procnone\} \subseteq s \cup \{procnone\}$ *y*
$w \cup \{procnone\} = ((s \cup \{procnone\}) \cup add(a'_1)) \setminus del(a'_1)$
$\Rightarrow pre(a_1) \subseteq s$ *y* $w \cup \{procnone\} = ((s \cup \{procnone\}) \cup add(a_1)) \setminus del(a_1)$
$\Rightarrow pre(a) \subseteq s$ *y* $w \cup \{procnone\} = ((s \cup add(a)) \setminus del(a)) \cup \{procnone\}$
$\Rightarrow pre(a) \subseteq s$ *y* $w = (s \cup add(a)) \setminus del(a)$
$\Rightarrow s \xrightarrow{a} w$

Caso $k > 1$*: debo ver que* $T(s) \xrightarrow{a'_1 \dots a'_k} T(w) \Rightarrow s \xrightarrow{a} w$.

$T(s) \xrightarrow{a'_1 ... a'_k} T(w)$
$T(s) \xrightarrow{a'_1 ... a'_{k-1}} s'_{k-1} \xrightarrow{a'_k} T(w)$
$pre_{i \leq k-1}(a_i) \subseteq s$ *y* $s'_{k-1} = (s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)) \cup \{do^{a'_k}\}$ ***por Lema 1***

$pre(a'_k) \subseteq s'_{k-1}$
$pre(a_k) \cup \{do^{a'_k}\} \subseteq (s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)) \cup \{do^{a'_k}\}$
$pre(a_k) \subseteq s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)$
$pre(a_k) \subseteq s \cup add_{i \leq k-1}(a_i)$ ***por Proposición 1***
$pre(a_k) \subseteq s$ ***por Proposición 1***

Como $pre_{i \leq k-1}(a_i) \subseteq s$ *y* $pre(a_k) \subseteq s$, *entonces* $pre_{i \leq k}(a_i) \subseteq s$.
$\bigcup pre_{i \leq k}(a_i) \subseteq s$
$pre(a) \subseteq s$
$s \xrightarrow{a} v$ *y* $v = (s \cup add(a)) \setminus del(a)$

Resta ver que $T(v) = T(w)$.
$T(w) = (s'_{k-1} \cup add(a'_k)) \setminus del(a'_k)$
$T(w) = (((s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)) \cup \{do^{a'_k}\}) \cup add(a'_k)) \setminus del(a'_k)$
$T(w) = (((s \cup_{i \leq k-1} add(a_i) \setminus_{i \leq k-1} del(a_i)) \cup \{do^{a'_k}\}) \cup (add(a_k) \cup \{procnone\}))$
$\setminus (del(a_k) \cup \{do^{a'_k}\})$
$T(w) = (s \cup_{i \leq k} add(a_i) \setminus_{i \leq k} del(a_i)) \cup \{procnone\}$
$T(w) = ((s \cup add(a)) \setminus del(a)) \cup \{procnone\}$
$T(w) = v \cup \{procnone\}$
$T(w) = T(v)$

Lema 4. *Sea* A *un dominio,* $s, w \in \Sigma_A$, $\overline{a'} \in Split_\alpha(A)^*$ *y* T, t *las funciones de traducción definidas en el Teorema 2.*

Si $T(s) \xrightarrow{\overline{a'}} T(w)$, *entonces* $\overline{a'} = t(\overline{a})$ *para algún* $\overline{a} \in A^*$.

Para la demostración de este lema, necesitamos primero agregar algunos conceptos nuevos.
$First = \{[t(a)]_1 : a \in A\}$, el conjunto de todas las primeras partes de acciones de A.
$Last = \{[t(a)]_{|t(a)|} : a \in A\}$, el conjunto de todas las últimas partes de acciones de A.
Sabemos que por definición de $Split_\alpha(A)$, si $a' \in Split_\alpha(A)$ entonces $a' \in t(a) = a'_1 ... a'_k$, para algún $a \in A$, por lo tanto, podemos definir la noción de sucesores entre las acciones de $Split_\alpha(A)$:
$suc : Split_\alpha(A) \to \mathcal{P}(Split_\alpha(A))$
$suc(a') = First$ si $a' \in Last$
$suc(a') = \{a'_{i+1}\}$ si $a' \notin Last$ y $a' = a'_i$.

De la definición de la función Γ se desprende que si $a' \in First$, entonces $\{procnone\} \in pre(a')$, mientras que si $a' \notin First$, entonces $\{do^{a'}\} \in pre(a')$.
Sea $t(A^*) = \{t(\overline{a}) : \overline{a} \in A^*\}$. Se puede ver que $\overline{a'} \in t(A^*) \setminus \varepsilon$ sii $[\overline{a'}]_1 \in First$, $[\overline{a'}]_{i+1} \in suc([\overline{a'}]_i)$ para todo $i = 1...|\overline{a'}|$-1, $[\overline{a'}]_{|\overline{a'}|} \in Last$. Ahora sí pasemos a la prueba del Lema 4.

Prueba 4. *Caso* $\overline{a'} = \varepsilon$: $t(s) \xrightarrow{\varepsilon} t(w)$, *luego* $\varepsilon = t(\varepsilon)$ *y* $\varepsilon \in A^*$.

Caso $\overline{a'} \neq \varepsilon$ *: por reducción al absurdo.*

Supongamos $T(s) \xrightarrow{\overline{a'}} T(w)$ *y* $\forall \overline{a} \in A^*$*:* $\overline{a'} \neq t(\overline{a}) = t(a_1...a_n)$*. Como* $\overline{a'} \neq t(\overline{a})$ *para todo* $\overline{a} \in A^*$*, entonces* $\overline{a'} \notin t(A^*)$*, luego* $\overline{a'} \notin t(A^*) \setminus \varepsilon$*, por lo tanto,* $[\overline{a'}]_1 \notin First$ *ó* $[\overline{a'}]_{i+1} \notin suc([\overline{a'}]_i)$ *para algún i=1...*$|\overline{a'}|$*-1 ó* $[\overline{a'}]_{|\overline{a'}|} \notin Last$*. Veamos estas tres posibilidades.*

Subcaso 1: $[\overline{a'}]_1 \notin First$*.*
Como $[\overline{a'}]_1 \notin First$*, entonces* $\overline{a'} = a'xs'$*, con* $a' \in Split_\alpha(A)$*,* $a' \notin First$ *y* $xs' \in Split_\alpha(A)^*$*.*
$T(s) \xrightarrow{a'xs'} T(w)$
$T(s) \xrightarrow{a'} v' \xrightarrow{xs'} T(w)$
$a' \notin First$
$\{do^{a'}\} \in pre(a')$
$T(s) = s \cup \{procnone\}$
$\{do^{a'}\} \notin T(s)$*, por lo tanto,* a' *no es ejecutable en* $T(s)$*. ABSURDO*

Subcaso 2: $[\overline{a'}]_{i+1} \notin suc([\overline{a'}]_i)$ *para algún i=1...*$|\overline{a'}|$*-1.*
Como $[\overline{a'}]_{i+1} \notin suc([\overline{a'}]_i)$ *para algún i=1...*$|\overline{a'}|$*-1, entonces* $\overline{a'} = xs'a'b'ys'$*, con* $a', b' \in Split_\alpha(A)$*,* $b' \notin suc(a')$*,* $xs', ys' \in Split_\alpha(A)^*$*.*
$T(s) \xrightarrow{xs'a'b'ys'} T(w)$
$T(s) \xrightarrow{zs'} v'_1 \xrightarrow{a'_1...a'_q} v'_2 \xrightarrow{b'} v'_3 \xrightarrow{ys'} v'_4$ *con* $a'_q = a'$ *y* $xs' = zs'a'_1...a'_{q-1}$ *y* $a'_1...a'_{q-1}$ *tramo inicial de* $t(a)$ *para algún* $a \in A^*$ *y* $zs' = t(\overline{z})$ *para algún* $\overline{z} \in A^*$
$T(s) \xrightarrow{t(\overline{z})} T(v_1) \xrightarrow{a'_1...a'_q} v'_2 \xrightarrow{b'} v'_3$ *por Corolario 1*
$T(v_1) \xrightarrow{a'_1...a'_q} v'_2 \xrightarrow{b'} v'_3$

Supongamos $a'_q \notin Last$*.*
$a'_1...a'_q$ *es tramo inicial propio de t(a).*
$v'_2 = (v_1 \cup_{i \leq q} add(ai) \setminus_{i \leq q} del(ai)) \cup \{do^{a'_{q+1}}\}$ *por Lema 1*
Si $b' \in First \Rightarrow \{procnone\} \in pre(b')$*, pero* $\{procnone\} \notin v'_2 \Rightarrow b'$ *es no ejecutable en* v'_2*. ABSURDO.*
Si $b' \notin First \Rightarrow \{do^{b'}\} \in pre(b')$*, pero* $\{do^{b'}\} \notin v'_2$*, pues* $\{do^{a'_{q+1}}\} = \{do^{suc(a'_q)}\} \neq \{do^{b'})\}$ *porque* $b' \notin suc(a'_q) \Rightarrow b'$ *es no ejecutable en* v'_2*. ABSURDO*

Supongamos $a'_q \in Last$*.*
$a'_1...a'_q$ *es tramo incial no propio de t(a), es decir, t(a) =*$a'_1...a'_q$*.*
$b' \notin suc(a'_q) = First \Rightarrow \{do^{b'}\} \in pre(b')$
Teniamos que $T(v_1) \xrightarrow{a'_1...a'_q} v'_2$
$T(v_1) \xrightarrow{t(a)} v'_2$
$T(v_1) \xrightarrow{t(a)} T(v_2)$ *con* $v_2 \in \Sigma_A$*, por Lema 2*
$T(v_1) \xrightarrow{a'_1...a'_q} T(v_2)$

$T(v_1) \xrightarrow{a'_1 \ldots a'_q} T(v_2) \xrightarrow{b'} v'_3$
$T(v_2) = v_2 \cup \{procnone\}$
$\{do^{b'}\} \notin T(v_2)$, *por lo tanto,* b' *es no ejecutable en* $T(v_2)$. ***ABSURDO.***

Subcaso 3: $[\overline{a'}]_{|\overline{a'}|} \notin Last$.
Como $[\overline{a'}]_{|\overline{a'}|} \notin Last$, *entonces* $\overline{a'} = xs'a'$, *con* $a' \in Split_\alpha(A)$, $a' \notin Last$ *y* $xs' \in Split_\alpha(A)^*$.
$T(s) \xrightarrow{xs'a'} T(w)$
$T(s) \xrightarrow{zs'} v'_1 \xrightarrow{a'_1 \ldots a'_q} T(w)$ *con* $a'_q = a'$ *y* $xs' = zs'a'_1 \ldots a'_{q-1}$ *y* $a'_1 \ldots a'_{q-1}$ *tramo inicial propio de* $t(a)$ *para algún* $a \in A^*$ *y* $zs' = t(\overline{z})$ *para algún* $\overline{z} \in A^*$
$T(s) \xrightarrow{t(\overline{z})} v'_1 \xrightarrow{a'_1 \ldots a'_q} T(w)$
$T(s) \xrightarrow{t(\overline{z})} T(v_1) \xrightarrow{a'_1 \ldots a'_q} T(w)$ *con* $v_1 \in \Sigma_A$
$T(v_1) \xrightarrow{a'_1 \ldots a'_q} T(w)$
$T(w) = (v_1 \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}$ *por Lema 1*
$w \cup \{procnone\} \neq (v_1 \cup_{i \leq q} add(a_i) \setminus_{i \leq q} del(a_i)) \cup \{do^{a'_{q+1}}\}$
$T(w) \neq T(w)$. ***ABSURDO.***