TEO: Una herramienta para la optimización de la probabilidad de ejecución de casos de prueba en tiempo real

Trabajo Especial de Licenciatura en Cs. de la Computación

Autor: Gabriel Leonardo Miretti

Directores: Pedro D'Argenio
Nicolás Wolovick

17 de diciembre de 2010

Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba
Argentina

.

Resumen

En las pruebas de sistemas de tiempo real, y en particular los que tienen comportamiento estocástico, es muy importante que éstas se ejecuten con la mayor probabilidad posible. El objetivo de este trabajo final es implementar de forma eficiente una solución al problema de optimización de la probabilidad de ejecución de casos de prueba de un modelo de sistema de tiempo real con salidas estocásticas y entradas controladas por el usuario. El tiempo en el que ocurren dichas entradas será tal que maximize la probabilidad de ocurrencia del caso de prueba en el modelo. La herramienta se basa en la equivalencia entre este problema y la maximización del volumen seccional de un politopo convexo. Esta implementación es complementaria a una investigación reciente de Wolovick, D'Argenio, y Qu (reportada en ICST 2009) sobre este problema. En particular, el desarrollo de la implementación permitió evidenciar y corregir algunos problemas en dicho trabajo.

Clasificación ACM 1998:
D.2.4 (Software Engineering - Software Verification - Formal Methods)
D.2.5 (Software Engineering - Testing and Debugging)
Palabras Claves:
Métodos formales, Software Testing, Sistemas estocásticos, Sistemas en tiempo real, Derivación de casos de pruebas, Volumen seccional, Volitopo convexo

Agradecimientos

- A mis viejos por darme la posibilidad de estudiar y desarrollarme como persona y profesional, espero recompensar algún día todo el amor y apoyo que me dieron. A mi hermano por ser mi socio de toda la vida. Y a toda mi familia, abuela, tíos, primos, por alentarme en todas las circunstancias y siempre pedirme que dé lo máximo en todo lo que emprenda.
- A mi amor, Mariana, por darle luz a mis días más oscuros, por apoyarme y alentarme a que me reciba de una buena vez. No lo hubiera logrado sin vos, te amo y gracias por hacerme tan feliz.
- A mis directores, Pedro D'Argenio y Nicolás Wolovick por confiar en mí para realizar este trabajo y por su eterna predisposición.
- A la banda del tío, por hacer de la facultad una experiencia mucho más placentera y enriquecedora que continuaremos toda la vida.
- A la facultad, por dar la excelente capacitación profesional y la predisposición de todos, docentes, ayudantes, personal de la Biblioteca, Departamento de Alumnos, etc. También al Estado Nacional y todos los argentinos por seguir apostando a la educación universitaria pública y gratuita como motor del desarrollo económico, productivo y social del país.
- A la comunidad de software libre no sólo por brindarme las herramientas que me permiten desarrollarme profesionalmente, en particular la realización de este trabajo, sino también por defender la libertad de los usuarios para ejecutar, copiar, distribuir, estudiar y/o, modificar el software.

Índice general

Capítulo 1

Introducción

1.1. Motivación

Testear software engloba un conjunto de actividades que tienen el objeto de evaluar la calidad del producto, identificando defectos y problemas. Más precisamente, consiste en verificar *dinámicamente* el comportamiento de un programa a través de un grupo *finito* de casos de prueba, debidamente *seleccionados* del, típicamente infinito, ámbito de ejecuciones, todo en relación al comportamiento *esperado*.

El concepto detrás del *testing basado en modelos* es usar modelos que expliciten el comportamiento esperado del sistema para generar y realizar la prueba de software. Las trazas de estos modelos son interpretadas como casos de prueba para la implementación, indicando las entradas a proveer y las salidas a esperar. La selección de un conjunto finito de trazas finitas del total de trazas del modelo, se determina de acuerdo a un criterio de cobertura particular, ya que usualmente no es posible generar conjuntos de prueba exhaustivos. Durante la ejecución, la implementación (o SUT, system under test) es alimentada con las entradas y la salida de esta es comparada con la del modelo.

Pensando en sistemas de tiempo real, las entradas no sólo están compuestas por las entradas propiamente dichas sino también por el momento en que éstas deben realizarse. Además si estamos ante la presencia de probabilismo, la ejecución de una traza particular del sistema no es certera, existe una probabilidad de que se ejecute y esta no sólo depende del orden de las entradas sino también de los tiempos en los que éstas se realicen.

Si la derivación, la ejecución y la evaluación de las pruebas son actividades costosas, además uno espera que la ejecución de las pruebas, entre otras cosas, sea eficiente. Por lo tanto, es necesario incrementar la probabilidad de éxito de la ejecución de un caso de prueba, donde por "éxito" queremos decir "que la prueba ejecute la traza esperada". Para eso la derivación del caso de prueba no sólo debe indicar entradas a realizar y salidas esperadas, sino también los tiempos óptimos de las entradas.

`TEO` pretende implementar una técnica que completa la derivación de un caso de prueba particular con los tiempos de entrada que dan la mayor probabilidad de éxito posible en la ejecución y cuantificar esta probabilidad. Esta técnica ha sido desarrollada por Pedro D'Argenio, Nicolás Wolovick y Hongyang Qu en el trabajo [42].

1.2. Objetivos

- Estudiar la eficiencia de la técnica propuesta en [42] mediante la realización de un prototipo de herramienta que la implemente.
- Estudiar las posibles optimizaciones a la técnica propuesta mediante modificaciones del prototipo anterior.
- Realizar estudios sobre casos de prueba con casos representativos.

1.3. Descripción general

El presente trabajo final consiste en la construcción de una herramienta que tome un caso de prueba de un sistema de tiempo real y provea los tiempos en los que deberá realizarse cada entrada de manera tal que se maximice la probabilidad de ejecución del mismo.

La técnica sugerida en [42] utiliza un modelo del sistema de tiempo real. En este modelo hay acciones de entrada (controladas) y de salida (no controladas): las entradas son completamente controladas por el usuario y las salidas son acciones temporizadas de acuerdo a una distribución uniforme. Asumiendo que los casos de prueba son obtenidos por alguna técnica ([41, 30, 40]), consideraremos a un caso de prueba como una secuencia de entradas y salidas que finalizan en un veredicto. La ejecución de casos de prueba puede ser vista como un juego entre el tester, que alimenta las entradas, y el sistema bajo prueba, que posee el rol opuesto de producir las salidas. Estas salidas pueden ser o no las esperadas en el caso de prueba. En este último caso el veredicto probablemente no será concluyente. La técnica deriva los tiempos óptimos de alimentación de cada entrada del caso de prueba al sistema bajo prueba de forma tal que el sistema es guiado con la mayor probabilidad al final del caso de prueba, donde el veredicto es llevado a cabo.

En el trabajo citado se muestra que este problema de optimización es equivalente al problema de maximización del volumen de una sección de un politopo convexo. Como ha sido señalado por sus autores la complejidad teórica de este último problema es exponencial. Aun así, dadas las características del problema original, los autores sostienen que en sistemas reales la complejidad debería ser significativamente menor y por lo tanto ser útil en términos prácticos. En este trabajo proveemos evidencia de esta hipótesis.

En la primer parte del trabajo, comenzaremos discutiendo qué es y cómo se realiza el testing basado en modelos de sistemas de tiempo real. En el Capítulo 3 discutiremos los fundamentos matemáticos necesarios para resolver este problema. En el capítulo siguiente describimos el modelo para sistemas estocásticos de tiempo real y su semántica. Discutimos qué es un caso de prueba, cómo determinar su probabilidad de ejecución y la equivalencia entre optimizar esta probabilidad y maximizar el volumen de un politopo conveco paramétrico en el Capítulo 5. También mostraremos la técnica propuesta para realizar esta maximización.

La segunda parte del trabajo, se inicia discutiendo el algoritmo general de la herramienta y cómo se implementó cada piso. La arquitectura, el diseño detallado y la metodología de desarrollo es discutida en el Capítulo 7.

En la última parte, presentaremos los resultados obtenidos de diferentes ejecuciones de la herramienta y las conclusiones obtenidas en este trabajo.

Parte I

Fundamentos teóricos

Capítulo 2

Testing de sistemas de tiempo real basado en modelos

¿Qué es testing de software?

El *testeo de software* es una actividad importante en el proceso de desarrollo de software. Engloba un conjunto de actividades que tienen el objeto de evaluar la calidad del producto, identificando defectos y problemas. Las pruebas del software consisten en verificar *dinámicamente* el comportamiento de un programa a través de un grupo *finito* de casos de prueba, debidamente *seleccionados* del, típicamente infinito, ámbito de ejecuciones, en relación al comportamiento *esperado*. [15]

Los siguientes aspectos caracterizan al testeo de software:

- Verificación dinámica: testear software siempre supone ejecutar el programa ingresando datos.
- Conjunto finito de pruebas: incluso en programas sencillos, teóricamente podría haber tantas pruebas que realizar, que hacer pruebas exhaustivas podría demorar meses o años.
- Pruebas seleccionadas: la diferencia esencial entre las diferentes técnicas de pruebas se encuentra en cómo se escoge el conjunto de pruebas.
- Comportamiento esperado: debería ser posible, aunque a veces no sea fácil, decidir si el resultado observado de la ejecución de un programa es aceptable o no.

¿Qué es y cómo se realiza la derivación de las pruebas?

De manera frecuente, el comportamiento esperado es definido con especificaciones de requerimientos informales e incompletas. Los analistas de testing usan estos documentos para obtener un entendimiento aproximado del comportamiento deseado. Podemos decir que construyen un modelo mental del sistema. Este modelo mental es entonces usado para derivar casos de prueba para la implementación. Este enfoque es implícito y desestructurado, no pone atención a los detalles y no es reproducible.

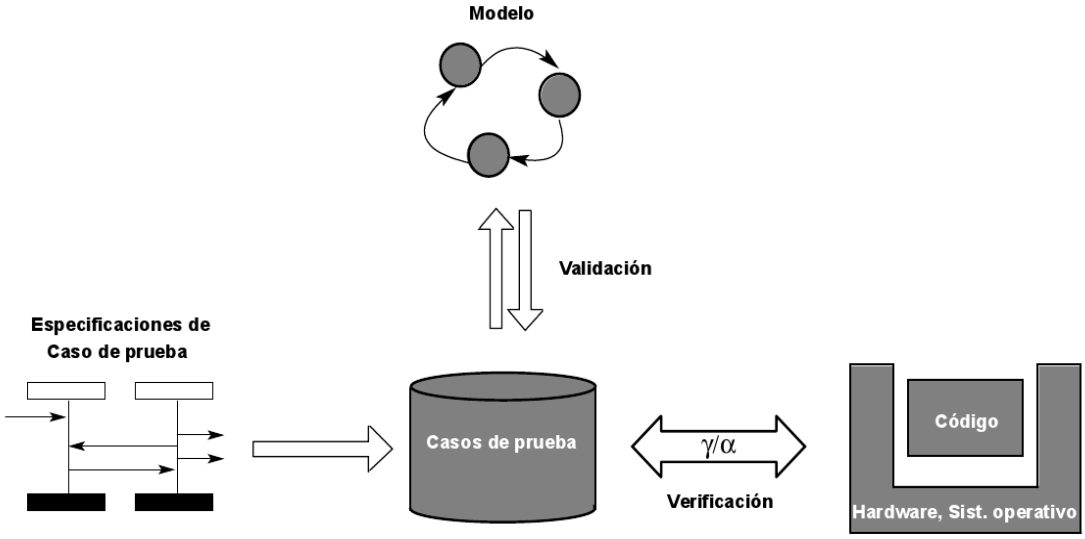
¿Qué es y cómo se realiza el testing basado en modelos?

Para contrarrestar las deficiencias de un enfoque informal de la derivación del testing, podemos utilizar el *testing basado en modelos.* Esta técnica prescribe la utilización de modelos que explicitan el comportamiento esperado del software para generar los casos de pruebas. Esto es debido a que las trazas de estos modelos son interpretadas como casos de prueba para la implementación indicando las entradas a proveer y las salidas esperadas de la implementación.

En el paso siguiente a la construcción del modelo, se seleccionan un conjunto de trazas del modelo de acuerdo a un criterio de cobertura particular. Esta etapa es conocida como especificación o derivación de los casos de prueba.

Antes de usarse como casos de pruebas, las trazas generadas también pueden ser verificadas manualmente para establecer si el modelo representa los requerimientos del sistema. Esta es una actividad de *validación*, encargada de verificar si un artefacto, el modelo en este caso, satisface los requerimientos de los usuarios reales.

Finalmente, en tiempo de ejecución las entradas son alimentadas a la implementación y la salida de la implementación se compara con la del modelo. Estas pruebas aumentarán la confianza que la implementación corresponde con el modelo, o probarán que no es así. La ejecución de los casos de prueba en el SUT es una actividad de *verificación*: indaga si la implementación se comporta adecuadamente.



Un punto esencial en la utilización de modelos es que estos son más abstractos o simples que la implementación. Por lo tanto son más fáciles de entender, validar, mantener, y más útiles para generar casos de prueba.

Los modelos de tipo autómata han sido ampliamente estudiados como base para la derivación de pruebas (por ejemplo [30, 41]).

¿Qué es y qué características poseen los sistemas en tiempo real?

Un sistema se denomina *de tiempo real* cuando su funcionalidad no sólo depende de los resultados lógicos de un cómputo sino también del tiempo en que esa computación se realiza [13]. Generalmente se encuentran en sistemas embebidos y probablemente desempeñando alguna funcionalidad crítica, donde la posibilidad de destrucción de información, bienes valiosos o la vida de personas está en juego. Por ejemplo, son usados en aviación, automóviles, sistemas de control de procesos industriales, etc. Por lo tanto, la corrección de estos sistemas es un tema fundamental para sus desarrolladores y usuarios. En este trabajo, nos enfocaremos en el testeo de este tipo de sistemas.

¿Qué desafíos y características tiene el testing de sistemas en tiempo real?

Las características particulares que presentan los sistemas de tiempo real, hacen más difíciles los problemas usuales del testing, ya que agregan problemas adicionales. Por ejemplo, el tiempo se incorpora como una variable continua no controlada del entorno de ejecución [38].

¿Cómo se usa testing basado en modelos en sistemas de tiempo real?

Las técnicas de testing basado en modelos en principio son aplicables al testeo de sistemas de tiempo real aún cuando el comportamiento temporal agrega una nueva dimensión del espacio de entradas. Pero la derivación de pruebas para sistemas de tiempo real ha resultado ser un duro problema a la hora de obtener un conjunto de pruebas exhaustivas [40]. Con esto en mente, estas técnicas no buscan exhaustividad sino que responden a un criterio particular de cobertura (ver por ejemplo [20, 32, 23, 17]).

¿Por qué es importante que las pruebas se ejecuten correctamente?

La ejecución de las pruebas de sistemas en tiempo real presenta la fuente de las diferencias más significativas en comparación con las pruebas de software que no es de tiempo real, parcialmente por las restricciones temporales, característica esencial de este tipo de sistemas, [38] y parcialmente por el no determinismo o probabilismo que suele caracterizarlos. Al ser de tiempo real, ahora no solo importa la secuencia de las entradas del caso de prueba sino también el tiempo de ocurrencia de las mismas. Al no ser determinista o siendo probabilista existe la posibilidad de que al querer reproducir un comportamiento, incluso proveyendo las mismas entradas, este comportamiento no se repita. Dado que la derivación, ejecución y evaluación de las pruebas son actividades costosas, uno espera que la ejecución de las pruebas sea eficiente. Además la reproducibilidad de las pruebas es muy importante para las *pruebas de regresión*. Estas son las pruebas que se realizan para probar si un cambio en el software no ha introducido nuevos errores en funcionalidades ya probadas.

¿Cómo modelamos los sistemas en este trabajo?

En este trabajo, asumiremos que un sistema de tiempo real es modelado como un conjunto de componentes en paralelo. Cada componente es modelado como un autómata controlado donde acciones (controladas) de entrada son dirigidas por el entorno y acciones (no controladas) de salida pueden ocurrir dentro de un intervalo de tiempo dado. La mejor opción posible para estimar el tiempo de ocurrencia de un evento cuando sólo un intervalo de tiempo es conocido, es asumiendo que este es uniformemente distribuido ya que la distribución uniforme es precisamente la distribución con máxima entropía soportada en un intervalo cerrado. También hay una consecuencia técnica de elegir la distribución uniforme: la función de densidad es una función constante y esto hace que el problema con el que trabajamos sea mucho más tratable. También consideraremos los casos de prueba modelados como una secuencia de entradas y salidas, finalizando en un veredicto [42].

¿Qué hacemos en este trabajo?

Por la presencia del no determinismo externo y el probabilismo interno, la ejecución del caso de prueba puede ser vista como un juego entre el tester, que realiza las entradas, y el sistema bajo pruebas, que juega el rol opuesto produciendo las salidas. Estas salidas pueden ser esperadas en los casos de prueba; o inesperadas, en cuyo caso el veredicto será probablemente inconcluso.

En este trabajo implementamos una herramienta que deriva el tiempo óptimo para realizar cada entrada del caso de prueba al SUT de modo que el sistema sea guiado con la máxima probabilidad hasta el final del caso de prueba donde se realiza el veredicto. Al incluir los tiempos de las entradas, el caso de prueba contará con los requisitos para realizar una ejecución, por lo cual podemos decir que se completa su derivación. A esta la llamaremos `TEO` por la abreviatura de "test execution optimizer", es decir "optimizador de ejecución de pruebas" en inglés.

En nuestra implementación haremos los cálculos de los tiempos de las entradas antes de la ejecución. Para eso utilizaremos el hecho de que el problema de calcular los tiempos óptimos de las entradas para maximizar la probabilidad de ejecución de un caso de prueba es equivalente a maximizar el volumen seccional de un politopo convexo. Este trabajo se basará en la técnica descripta por Wolovick, D'Argenio y Qu [42].

¿Qué no hacemos en este trabajo?

El objetivo de este trabajo no abarca la derivación completa de casos de prueba, sino completarlos con los mejores tiempos en lo que las entradas deben ser provistas. Al igual que en [42], asumiremos que la secuencia de entradas y salidas de los casos de pruebas son obtenidos de alguna forma (probablemente, de una abstracción) del modelo. En el mismo trabajo se citan al menos dos técnicas conocidas para la derivación de casos de pruebas sobre modelos más abstractos que el utilizado en este trabajo: *Input/Output labelled transition system* [41], y *observable nondeterministic finite state machine* [30]. Incluso cuando más

técnicas conocidas pueden generar estas secuencias, hay que considerar cuidadosamente los casos generados. Cuando el tiempo es ignorado por la técnica, pueden derivarse casos de prueba imposibles de ejecutar. Es de esperar que `TEO` detecte estas secuencias sin posibilidad de ser ejecutadas.

Otras técnicas [20, 32, 23, 16, 40] consideran el tiempo al realizar las derivaciones y no generan pruebas imposibles. Aun así la derivación de pruebas temporizadas puede volverse muy costosa si intenta ser exhaustiva[40]. Si bien el cálculo de probabilidad de ejecución podría realizarse sobre los casos de prueba generados por estas técnicas, no forma parte de este trabajo aunque sí da evidencia de que puede ser implementado.

Capítulo 3

Base matemática

3.1. Probabilidad

Dado un conjunto Ω y una colección $\mathcal{F}$ de subconjuntos de Ω, llamaremos a $\mathcal{F}$ una *σ-álgebra* si y sólo si $\Omega \in \mathcal{F}$ y $\mathcal{F}$ es cerrada bajo complemento y unión contable. Nosotros llamaremos a este par $(\Omega, \mathcal{F})$ *un espacio medible*. Cualquier conjunto $A \in \mathcal{F}$ es llamado medible.

Un *espacio topológico* es un conjunto X junto con T, una colección de subconjuntos de X, satisfaciendo los siguientes axiomas:

1. El conjunto vacío y X están en T.
2. La unión de cualquier colección de conjuntos T también se encuentra en T.
3. La intersección de cualquier colección finita de conjuntos T también se encuentra en T.

La colección T es llamada una *topología* sobre X. Los conjuntos en T son llamados conjuntos abiertos.

Un *conjunto de Borel* es cualquier conjunto en un espacio topológico que puede ser formado por conjuntos abiertos por las operaciones de unión contable, intersección contable y diferencia.

Para un espacio topológico T, la colección de todos los conjuntos de Borel de T, forman σ-álgebra de Borel de T que denotaremos por $\mathcal{B}(T)$. Por ejemplo, $\mathcal{B}(\mathbb{R}^+)$ denota el conjunto generado por todos los intervalos abiertos por derecha de $\mathbb{R}^+$. $\mathcal{B}(T)$ es la menor σ-álgebra que contiene todos los conjuntos abiertos.

Dada una σ-álgebra $\mathcal{A}$, una función $\mu : \mathcal{A} \to [-\infty, +\infty]$ es llamada σ-aditiva si para toda secuencia $A_1, A_2, ..., A_k, ...$, de conjuntos disjuntos en $\mathcal{A}$ vale que $\mu(\cup_{n=1}^{\infty} A_n) = \sum_{n=1}^{\infty} \mu(A_n)$

Dado un espacio medible $(\Omega, \mathcal{F})$, una función σ-aditiva $P : \mathcal{F} \to [0, 1]$ es llamada una *medida de probabilidad* si $P(\Omega) = 1$. La terna $(\Omega, \mathcal{F}, P)$ es un espacio medible probabilístico. Una función $f : (\Omega_1, \mathcal{F}_1) \to (\Omega_2, \mathcal{F}_2)$ es *medible* si $\forall A_2 \in \mathcal{F}_2, f^{-1}(A_1) \in \mathcal{F}_1$, i. e. la función inversa mapea medibles a medibles. Esta condición es satisfecha por una σ-álgebra de Borel sobre los reales si y sólo si $f^{-1}([a, b)) \in \mathcal{B}(\mathbb{R}^+)$, i. e. esto tiene que ser válido para los generadores, o sea, para los intervalos $[a, b)$ con $a, b \in \mathbb{Q}$. Una variable aleatoria X sobre un espacio de probabilidad $(\Omega, \mathcal{F}, P)$ es una función medible de Borel de Ω a $\mathbb{R}$. Por ejemplo, la función

identidad o cualquier función lineal es una variable aleatoria en el espacio de probabilidad $(\mathbb{R}, \mathcal{B}(\mathbb{R}), P)$. Dada una variable aleatoria X, la medida de probabilidad inducida por X es $P_X(B) \doteq P\{\omega \mid X(\omega) \in B\} = P(X^{-1}(B))$, donde $B \in \mathcal{B}(\mathbb{R})$ y $X^{-1}(B)$ es un conjunto medible ya que X es medible. Una función medible de Borel f es la *densidad de probabilidad* de la variable aleatoria X si $P_X(B) = \int_B f(x)dx$ para todo $B \in \mathcal{B}(\mathbb{R})$. Ejemplos de una función de densidad es la distribución uniforme en $[a, b]$, $f(x) = (\textbf{if } a \leq x \leq b \textbf{ then } (b-a)^{-1} \textbf{ else } 0)$. Un *vector aleatorio* X es una función medible de Ω a $\mathbb{R}^n$, a la cual las definiciones previas aplican y pueden considerarse como n variables aleatorias $(X_1, ..., X_n)$. Estas variables aleatorias se dicen independientes si $P\{X_1 \in B_1, ..., X_n \in B_n\} = P\{X_1 \in B_1\}...P\{X_n \in B_n\}$ y esto corresponde a la intuición de que el conocimiento de cualquier subconjunto de variables aleatorias no afecta las probabilidades del resto. Dado un vector $X = X_1, ..., X_n$ con funciones de densidad $f_1, ..., f_n$ por la definición previa y [8, Corolario 4.8.5] tenemos que:

$$P_X(B) = \int_B f_1(x_1)...f_n(x_n)dx_1...dx_n$$

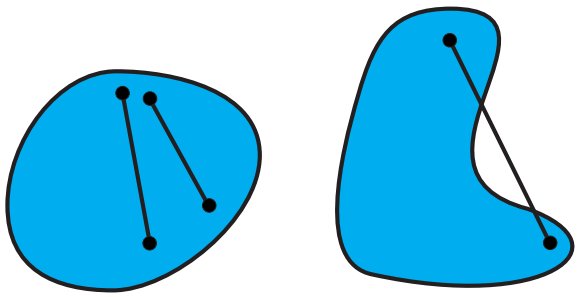
3.2. Geometría

Definición 3.1. Un subconjunto K del espacio euclídeo se llama *convexo* si para todo $x_1, x_2 \in K$ y $0 \leq \lambda \leq 1$ entonces $\lambda x_1 + (1-\lambda)x_2 \in K$. i. e. si contiene a todo segmento lineal que conecta a todos sus pares de puntos.

De manera opuesta,

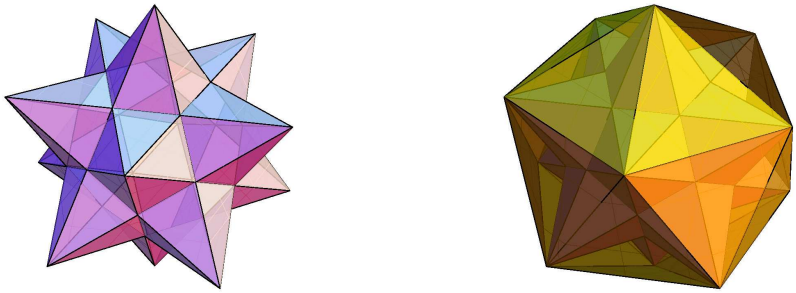
Definición 3.2. Un subconjunto K del espacio euclídeo se llama *cóncavo* si existen $x_1, x_2 \in K$ y $0 \leq \lambda \leq 1$ tal que $\lambda x_1 + (1-\lambda)x_2 \notin K$. i. e. si contiene un segmento lineal que conecta a un par de puntos propios, pero incluye puntos fuera de este.

En $\mathbb{R}^2$, podemos visualizar los conceptos en estos ejemplos:



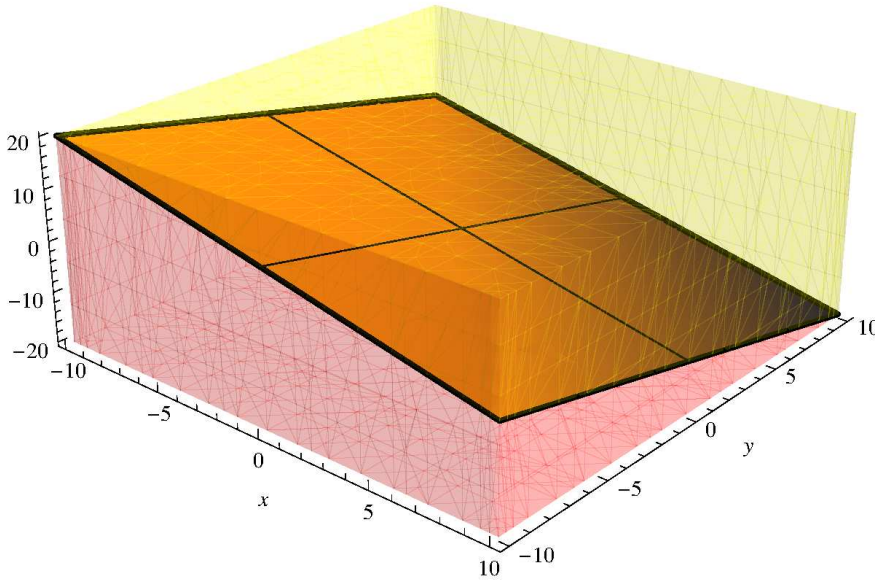
convexo *cóncavo*

El dodecaedro estrellado pequeño (a la izquierda) es un conjunto cóncavo en $\mathbb{R}^3$. La *envoltura convexa* de este, es decir la intersección de todos los conjuntos convexos que lo contienen, es el icosaedro (a la derecha).



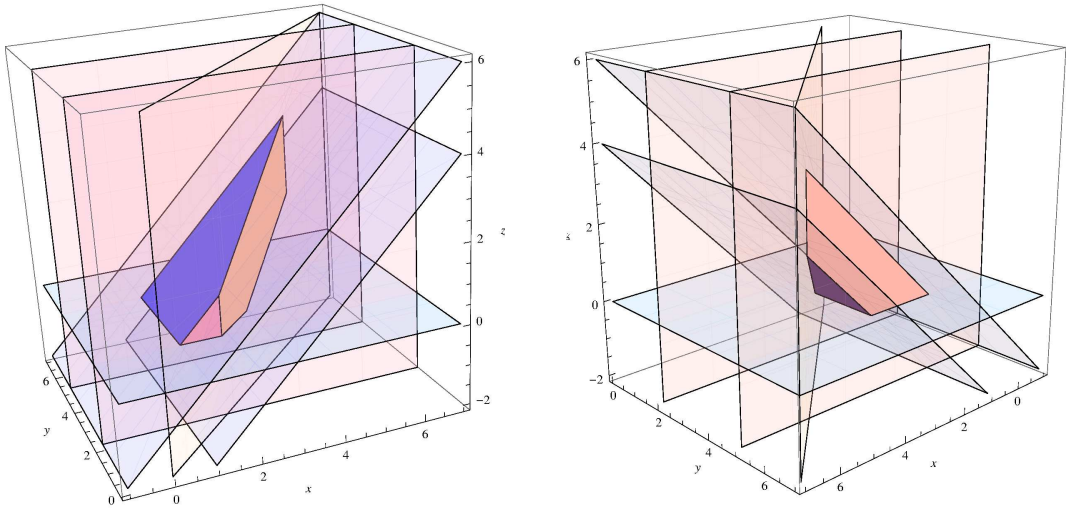
Definición 3.3. En $\mathbb{R}^n$ llamamos *semiespacio* a los conjuntos de puntos en una de las dos regiones en el cual un *hiperplano* divide el espacio completo, i. e. las soluciones de $a_1x_1 + ... + a_nx_n \leq b$.

Por ejemplo, en $\mathbb{R}^3$ el hiperplano $z = -x - y$ divide el espacio completo en los semiespacios $-x - y + z \geq 0$ (en amarillo) y $-x - y + z \leq 0$ (en rojo).



Definición 3.4. Un *politopo convexo* es un subconjunto acotado de $\mathbb{R}^n$ el cual puede ser definido de forma equivalente como soluciones de la desigualdad lineal $A\vec{x} \leq \vec{b}$ o la intersección finita de semiespacios.

Los politopos convexos son cerrados bajo intersecciones y proyecciones. Por ejemplo, podemos ver como 6 semiespacios en $\mathbb{R}^3$ definen un politopo convexo.

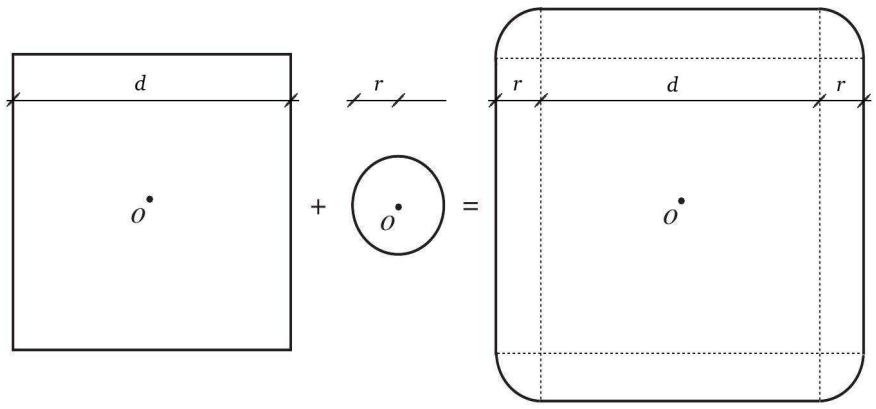


Definición 3.5. Dados dos politopos K_1, K_2 la *suma de Minkowski* es la suma de todos los vectores que pertenecen a cada politopo $K_1 + K_2 \doteq \{x_1 + x_2 \mid x_1 \in K_1, x_2 \in K_2\}$.

De forma similar podemos definir la multiplicación escalar

Definición 3.6. Dados un politopos K_2 y un escalar λ la *multiplicación escalar* de un politopo es el conjunto $\lambda K = \{\lambda x \mid x \in K\}$.

Por $Vol(K)$ denotaremos el *volumen del politopo* K.
En la siguiente figura vemos la suma de Minkowski de un cuadrado y un círculo.

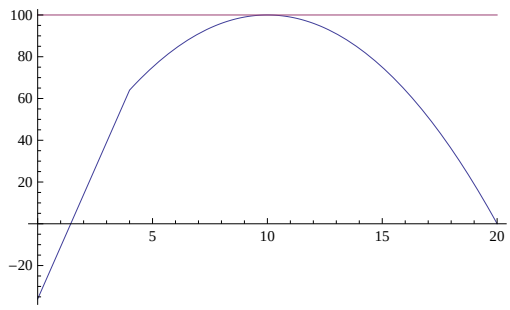


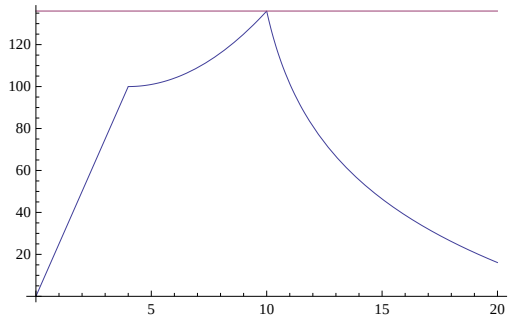
Una función convexa es una función continua cuyo valor en el punto medio de cada intervalo dentro de su dominio no supera a la media aritmética de los valores en los extremos de dicho intervalo. En términos formales:

Definición 3.7. Una función $f(x)$ a $\mathbb{R}$ se dice que es *convexa* si para todo par de puntos x_1 y x_2 en el dominio X y todo $t \in [0,1]$ vale: $f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$. Una función f se dice que es *cóncava* si $-f$ es convexa.

Una función $f(x)$ entre dos conjuntos ordenados es *unimodal* si para algún valor m (la *moda*), esta es monótona creciente para $x \leq m$ y monótona decreciente para $x \geq m$. Es este caso, el valor máximo de $f(x)$ es $f(m)$ y no hay otro máximo local.

A continuación observamos dos funciones unimodales continuas, notando que la función a la derecha además es cóncava.

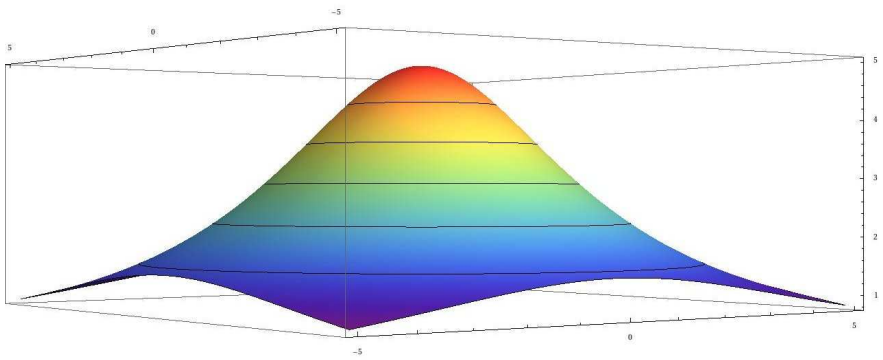




Este concepto es extendido a funciones medibles multidimensionales [25].

Definición 3.8. Si f es una función medible no negativa en $\mathbb{R}^d$, diremos que f es *unimodal* si los conjuntos de nivel $L(f,t) \doteq \{x \mid f(x) \geq t\}$ son convexos para todo $t \geq 0$.

Por ejemplo, la función $f : \mathbb{R}^2 \to \mathbb{R}$ con $f(x) = \frac{50}{x^2+y^2+10}$ es unimodal.



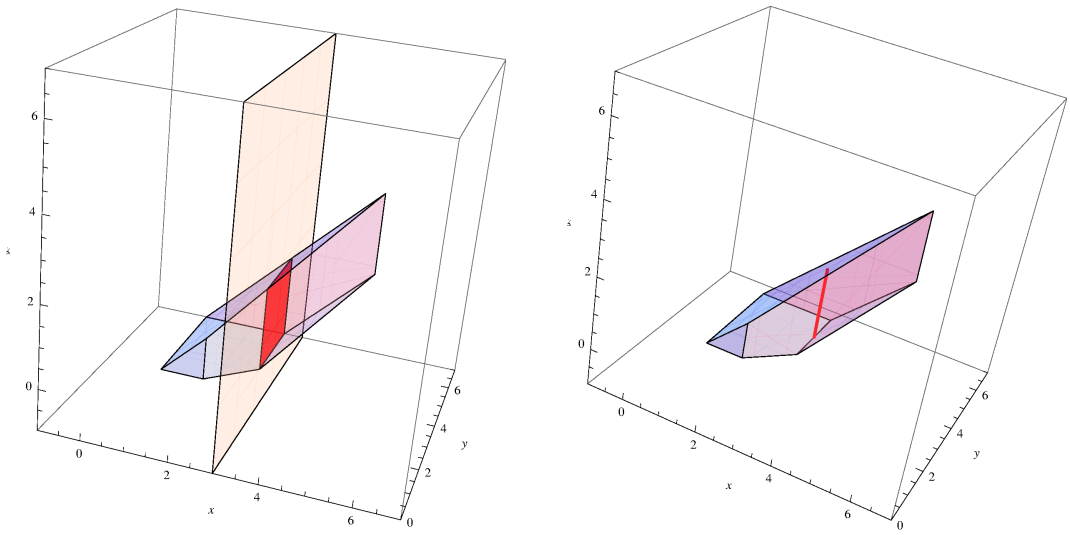
La convexidad de los conjuntos de niveles nos permite decir lo siguiente.

Proposición 3.9. En cada función unimodal, el máximo local coincide con el máximo global.

Definición 3.10. Dado un politopo convexo $K \subseteq \mathbb{R}^d$ y la función generadora de subespacio $H : \mathbb{R}^c \to \mathbb{R}^d$ con $c \leq d$, donde $H(x_1, ..., x_c) = \{(x_1, ..., x_c, x_{c+1}, ..., x_d) \mid x_{c+1}, ..., x_d \in \mathbb{R}\}$, definimos el *volumen seccional del politopo* $V_K(\vec{x}) \doteq Vol(K \cap H(\vec{x}))$.

Notar que si bien la función H llega a $\mathbb{R}^d$, el resultado de aplicar H es un volumen de dimensiones $d - c$.

Sea $K \subseteq \mathbb{R}^3$ un politopo convexo, en las siguientes figuras podemos ver las secciones al intersecar K con $H(3)$ y $H(3,4)$.



La desigualdad de Brunn-Minkowski [25] relaciona el volumen de dos cuerpos convexos:

Teorema 3.11. *Dados politopos convexos K, L en $\mathbb{R}^d$ y $\lambda \in (0,1)$ la siguiente desigualdad es válida: $Vol(\lambda K + (1-\lambda)L)^{1/d} \geq \lambda\, Vol(K)^{1/d} + (1-\lambda)\, Vol(L)^{1/d}$.*

Tendremos en cuenta la siguiente inclusión de conjuntos que involucra la suma de Minkowsky.

Corolario 3.12. Dados politopos convexos K, L en $\mathbb{R}^d$ con $0 \leq \lambda \leq 1$ la siguiente inclusión es válida: $K \cap H(\lambda\vec{x_1} + (1-\lambda)\vec{x_2}) \supseteq \lambda(K \cap H(\vec{x_1})) + (1-\lambda)(K \cap H(\vec{x_2}))$

El siguiente corolario es el resultado principal de este capítulo.

Corolario 3.13. El volumen seccional $V(\vec{x})$ de un politopo convexo es una función unimodal.

Demostración. Sea $\vec{x_1}, \vec{x_2} \in \mathbb{R}^c$ y $S_1 = K \cap H(\vec{x_1})$, $S_2 = K \cap H(\vec{x_2})$. Observar que S_1, S_2 son politopos convexos en $\mathbb{R}^{d-c}$. Recordemos que de acuerdo a 3.8, $V(\vec{x})$ es unimodal si los conjuntos de nivel $L(V,t)$ son convexos para todo $t \geq 0$. Ahora suponiendo que $0 \leq \lambda \leq 1$ y $\vec{x_1}, \vec{x_2} \in L(V,t)$, con $V(\vec{x_1}), V(\vec{x_2}) \geq t$, demostraremos que $V(\lambda\vec{x_1} + (1-\lambda)\vec{x_2}) \geq t$.

$$
\begin{aligned}
& V(\lambda\vec{x_1} + (1-\lambda)\vec{x_2})^{1/(d-c)} \\
= \quad & \{\text{De acuerdo a 3.10}\} \\
& Vol(K \cap H(\lambda\vec{x_1} + (1-\lambda)\vec{x_2}))^{1/(d-c)} \\
\geq \quad & \{\text{De acuerdo a 3.12}\} \\
& Vol(\lambda(K \cap H(\vec{x_1})) + (1-\lambda)(K \cap H(\vec{x_2})))^{1/(d-c)} \\
= \quad & \{\text{Definición de } S_1,\ S_2\} \\
& Vol(\lambda(S_1) + (1-\lambda)S_2))^{1/(d-c)} \\
\geq \quad & \{\text{De acuerdo a 3.11}\} \\
& \lambda Vol(S_1)^{1/(d-c)} + (1-\lambda)Vol(S_2)^{1/(d-c)} \\
= \quad & \{\text{Definición de } S_1,\ S_2\} \\
& \lambda Vol(K \cap H(\vec{x_1}))^{1/(d-c)} + (1-\lambda)Vol(K \cap H(\vec{x_2}))^{1/(d-c)} \\
= \quad & \{\text{De acuerdo a 3.10}\} \\
& \lambda V(\vec{x_1})^{1/(d-c)} + (1-\lambda)V(\vec{x_2})^{1/(d-c)} \\
\geq \quad & \{\text{Dado que } \vec{x_1}, \vec{x_2} \in L(V,t)\} \\
& \lambda t^{1/(d-c)} + (1-\lambda)t^{1/(d-c)} \\
= \quad & \{\text{Aritmética}\} \\
& t^{1/(d-c)}
\end{aligned}
$$

□

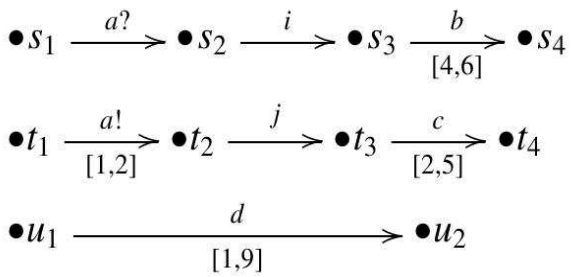
Capítulo 4

Autómata de entrada/salida temporizado estocástico

Definición 4.1. Un *autómata de entrada/salida temporizado estocásticamente* (STIOA, del inglés Stochastic Timed I/O Automaton) es una estructura $\mathcal{T} = (V, \Sigma)$ donde:

1. V es un conjunto finito de *variables de programa*, y
2. Σ es un conjunto finito de *transiciones* particionadas en dos conjuntos: el conjunto Σ_I de *transiciones de entrada* y el conjunto Σ_O de *transiciones de salida*.
   Cada transición $\alpha \in \Sigma$ incluye los siguientes componentes:
   - *a*) un predicado de primer orden $en(\alpha)$ sobre las variables V llamado *condición habilitante* de α
   - *b*) una *función de transformación* $F_\alpha : V \to V$ sobre las variables V, la cual es aplicada a ellas mismas si la transición α es tomada
   - *c*) para transiciones de salida $\alpha = \alpha_O \in \Sigma_O$, el intervalo $[\, l_{\alpha_O},\ u_{\alpha_O}]$ establece la cota inferior y superior de la duración del período durante el cual vale $en(\alpha_O)$ antes de que α_O pueda ser ejecutada, nosotros requerimos $l_{\alpha_O} < u_{\alpha_O}$. Por simplicidad, ambas cotas pertenecen a $\mathbb{N}$.

Ejemplo 4.2. Veamos un sistema que tiene tres componentes paralelas con dos transiciones de entrada i, j y cuatro de salida a, b, c, d. La transición de salida a es usada para sincronizar la habilitación de i y j, y para ese propósito fue dividida en emisor $a!$ y receptor $a?$ sincronizados.



Formalmente, llamaremos a este STIOA $M = (V, \Sigma)$ donde:

1. El conjunto de variables $V = s, t, u$ con $s, t \in \{1, 2, 3, 4\}$ y $u \in \{1, 2, 3\}$.
2. El conjunto de transiciones $\Sigma = \Sigma_I \cup \Sigma_O$ es formado por $\Sigma_I = \{i, j\}$ y $\Sigma_O = \{a, b, c, d\}$.
3. Las condiciones habilitantes de las transiciones son:

$$\begin{aligned} en(a) &\doteq s = 1 \wedge t = 1 \\ en(b) &\doteq s = 3 \\ en(c) &\doteq t = 3 \\ en(d) &\doteq u = 1 \\ en(i) &\doteq s = 2 \\ en(j) &\doteq t = 2 \end{aligned}$$

4. Las funciones de transformación son:

$$\begin{aligned} F_a(s,t,u) = (s+1, t+1, u) \quad & si \quad en(a) \\ F_b(s,t,u) = (s+1, t, u) \quad & si \quad en(b) \\ F_c(s,t,u) = (s, t+1, u) \quad & si \quad en(c) \\ F_d(s,t,u) = (s, t, u+1) \quad & si \quad en(d) \\ F_i(s,t,u) = (s+1, t, u) \quad & si \quad en(e) \\ F_j(s,t,u) = (s, t+1, u) \quad & si \quad en(f) \\ F_\alpha(s,t,u) = (s,t,u) \quad & \quad cc \end{aligned}$$

5. Las cotas de ejecución para las transiciones de salida α_O son l_{α_O} para la cota inferior y u_{α_O} para la superior:

$$\begin{aligned} l_a = 1 \quad , \quad u_a = 2 \\ l_b = 4 \quad , \quad u_b = 6 \\ l_c = 2 \quad , \quad u_c = 5 \\ l_d = 1 \quad , \quad u_d = 9 \end{aligned}$$

Cada transición de salida α_O está asociada con un *reloj regresivo* $cl(\alpha_O)$. Cada vez que α_O se habilita, o sea $en(\alpha_O)$ pasa a ser válida, $cl(\alpha_O)$ se establece en un valor en el intervalo $[\ l_{\alpha_O},\ u_{\alpha_O}]$. Entonces su valor comienza decrecer. Notar que los relojes no son discretos, el valor de cada reloj pertenece a $\mathbb{R}$. Una vez que es igual a cero, α_O es ejecutada. Si α_O es deshabilitada, deja de valer $en(\alpha_O)$, antes de que su reloj alcance cero, el reloj deja de correr y vuelve a cero (notar que α_O no es ejecutada). También hay un *reloj global* incremental $cl(gt)$ por sistema. Cuando el sistema empieza a ejecutarse, este comienza a correr y nunca para. Un estado s de $\mathcal{T}$ está conformado por el vector asignación $V(s)$ de las variables de programa y los valores de los relojes de las transiciones de salida $cl(\alpha)(s)$ y el global $cl(gt)(s)$. Para cada

$\alpha_O \in \Sigma_O$, $cl(\alpha_O)(s)$ (respectivamente $cl(gt)(s)$) denota la lectura de $cl(\alpha_O)$ (respectivamente $cl(gt)$) en el estado s, y por cada $\alpha \in \Sigma_O \cup \Sigma_I$, $s \vDash en(\alpha)$ representa que la condición $en(\alpha)$ de α vale en el estado s, i. e., α está habilitado en s.

En este modelo son diferentes las transiciones de entrada y salida, obteniendo un sistema internamente probabilístico y externamente no determinístico.

Las transiciones de salida no son gobernadas por el ambiente y el tiempo de ocurrencia de una transición de salida $\alpha_O \in \Sigma_O$ respeta una distribución de probabilidad cuyo conjunto de salida es el intervalo $[\, l_{\alpha_O},\ u_{\alpha_O}]$. En el resto del trabajo, tomaremos que esta distribución es uniforme en dicho intervalo.

En cambio, las transiciones de entrada son controladas por el ambiente. El preciso momento en el cual una transición de entrada toma lugar es elegido de forma no determinística por el ambiente, respetando siempre la condición habilitante. Entonces no obedece ninguna distribución de probabilidad inherente al modelo.

Definimos a una ejecución probabilística como un secuencia alternante de pasos de tiempo y ejecuciones de transición comenzando en un estado inicial s_0.

Definición 4.3. Una *ejecución probabilística* de un sistema $\mathcal{T}$ es una secuencia finita de la forma $s_0 g_1 \alpha_1 s_1 g_2 \alpha_2 s_2 g_3 ... s_{n-1} g_n \alpha_n s_n$, donde s_i,g_i son estados y α_i son transiciones. El estado s_0 es el estado inicial de $\mathcal{T}$. Si una transición $\beta \in \Sigma_O$ está habilitada $s_0 \vDash en(\beta)$, entonces $cl(\beta)(s_0)$ es un valor elegido al azar de $[l_\beta, u_\beta]$. El par adyacente s_i, g_{i+1} representa un *paso de tiempo*, por lo tanto $V(s_i) = V(g_{i+1})$, y donde todos los relojes habilitados se descuentan uniformemente, entonces $cl(\alpha_{i+1})(s_i) - cl(\alpha_{i+1})(g_{i+1}) = cl(gt)(g_{i+1}) - cl(gt)(s_i)$, donde $\alpha_i \in \Sigma_O$ y $s_i, g_{i+1} \vDash en(\alpha_i)$. La tripla $g_i \alpha_i s_i$ representa la *ejecución de una transición*, por lo tanto α_i tiene que estar habilitado en g_i, i. e., $g_i \vDash en(\alpha_i)$, si $\alpha_i \in \Sigma_O$, y el valor de su reloj debe ser 0, esto es $cl(\alpha_i)(g_i) = 0$. La ejecución de la transición es instantánea es decir $cl(gt)(g_i) = cl(gt)(s_i)$, y el estado cambia de acuerdo a $V(s_i) = F_{\alpha_i}(V(g_i))$. Si una ejecución de transición $\beta \in \Sigma_O$ pasa a estar habilitada por la ejecución de α_i tal que $g_i \nvDash en(\beta)$ y $s_i \vDash en(\beta)$, o $\beta = \alpha_i$ vuelve a estar habilitada otra vez $s_i \vDash en(\beta)$, entonces $cl(\beta)(s_i)$ es un valor elegido al azar de $[l_\beta, u_\beta]$. En el caso de que una transición $\gamma \in \Sigma_O$ sea deshabilitada por la ejecución de α_i tal que $g_i \vDash en(\gamma)$ y $s_i \nvDash en(\gamma)$, su reloj se reinicia, i. e., $cl(\gamma)(s_i) = 0$. Los relojes de otras transiciones permanecen sin cambios en s_i y g_i.

Definición 4.4. Un *camino* es una secuencia finita de acciones $\sigma = \alpha_1 ... \alpha_n$. Es *consistente* con una ejecución ρ si es obtenida eliminando todos los estados de ρ.

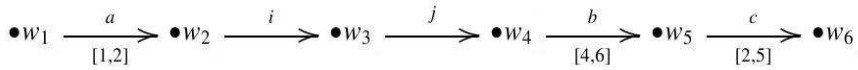
Ejemplo 4.5. A continuación vemos la composición paralela del sistema de 4.2 para el camino $\rho = aijbc$ de una ejecución $s_0 g_1 a s_1 g_2 i s_2 g_3 j s_3 g_4 b s_4 g_5 c s_5$.

Los estados w_i denotan a un estado entre s_{i-1} y g_i y que contiene los mismos valores en las variables, o sea $V(w_i) = V(s_{i-1}) = V(g_i)$. Podemos notar que múltiples ejecuciones pueden realizar el mismo camino pero variando los momentos en que las transiciones ocurren.

Denotaremos con $val(i) = (s, t, u)$ la terna con los valores de las variables s, t y u para el estado w_i. En este ejemplo vemos que:

$$\begin{aligned} val(1) &= (1,1,1) \\ val(2) &= (2,2,1) \\ val(3) &= (3,2,1) \\ val(4) &= (3,3,1) \\ val(5) &= (4,3,1) \\ val(6) &= (4,4,1) \end{aligned}$$

Debajo observamos la representación gráfica de la ejecución.



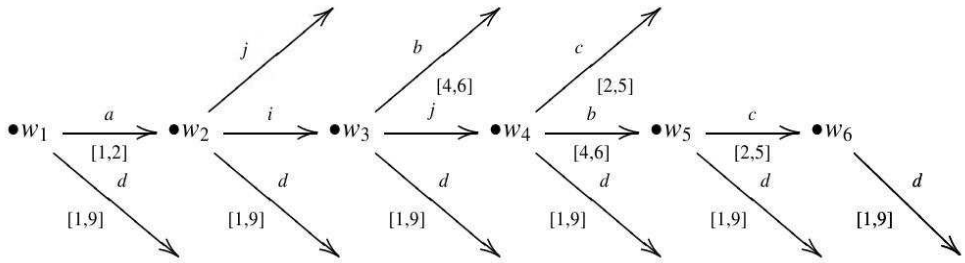
Cuando el sistema alcanza un estado, se define un conjunto de transiciones que están habilitadas. En este punto, comienza una carrera si hay más de una transición habilitada. Por lo tanto la probabilidad de tomar una transición en particular es una función de todas las transiciones habilitadas.

Definición 4.6. El *intervalo habilitante* de una transición α (no necesariamente en ρ) con respecto a la ejecución $\rho = s_0 g_1 \alpha_1 s_1 g_2 \alpha_2 s_2 g_3 ... s_{n-1} g_n \alpha_n s_n$ es $[\mathsf{start}(\alpha), \mathsf{stop}(\alpha)]$, donde $\mathsf{start}(\alpha) = min\{j \mid 0 \leq j \leq (n-1) \wedge s_j \vDash en(\alpha)\}$ y $\mathsf{stop}(\alpha) = max\{j \mid 1 \leq j \leq n \wedge g_j \vDash en(\alpha)\}$. Si la transición nunca es habilitada en ρ, su intervalo habilitante es *vacío*.

Una transición es una *transición participante* de una ejecución ρ si su intervalo habilitante con respecto a ρ es no vacío.

Ejemplo 4.7. Continuando con el ejemplo 4.5 podemos ver las transiciones participantes en ρ. Los intervalos habilitantes respectivos serían :

$$\begin{aligned} \mathsf{start}(a) = 0 \quad &, \quad \mathsf{stop}(a) = 1 \\ \mathsf{start}(b) = 2 \quad &, \quad \mathsf{stop}(b) = 4 \\ \mathsf{start}(c) = 3 \quad &, \quad \mathsf{stop}(c) = 5 \\ \mathsf{start}(d) = 0 \quad &, \quad \mathsf{stop}(d) = 5 \\ \mathsf{start}(i) = 1 \quad &, \quad \mathsf{stop}(i) = 2 \\ \mathsf{start}(j) = 1 \quad &, \quad \mathsf{stop}(j) = 3 \end{aligned}$$



Por definición α_i en ρ posee un intervalo habilitante no vacío que incluye i. Consideramos todas las transiciones participantes en un camino como diferentes, renombrando cada ocurrencia cuando sea necesario para cumplir con esta restricción. Entonces podemos ver que el intervalo habilitante de α_i en ρ es continuo, i. e. $s_{\mathsf{start(i)}} \vDash en(\alpha_i)$, $g_{\mathsf{stop(i)}} \vDash en(\alpha_i)$ y $\forall j : j \in [\mathsf{start}(i) + 1, \mathsf{stop}(i) - 1]: s_j \vDash en(\alpha_i) \wedge g_{\mathsf{j}} \vDash en(\alpha_i)$.

Capítulo 5

Optimización de probabilidad de ejecución de prueba

5.1. Probabilidad de ejecución de un caso de prueba

Primero definimos un *caso de prueba* como un camino que finaliza en una transición de salida, i.e. dado $\sigma = \alpha_1\alpha_2...\alpha_n$, entonces $\alpha_n \in \Sigma_O$. Dado un caso de prueba corriendo bajo la ejecución de $\rho = s_0g_1\alpha_1s_1g_2\alpha_2s_2g_3...s_{n-1}g_n\alpha_ns_n$, sea $[\rho] = \{\alpha_1, ..., \alpha_n\}$ el conjunto de transiciones en el caso de prueba y $[\rho]' = \{\alpha_{n+1}, ..., \alpha_m\}$ el conjunto de transiciones que han estado habilitadas en algún momento a lo largo de ρ pero que no pertenecen a $[\rho]$, i. e. las que no son ejecutadas en $[\rho]$. No consideraremos las transiciones de entrada que son habilitadas durante ρ pero lo desvían de su ejecución. Dichas entradas podrían hacer que el caso de prueba alcance un veredicto no concluyente, algo que uno puede evitar ya que las entradas son controlables.

Sea $[\rho]_I = \{I_1, ..., I_k\} \subseteq [\rho] \cup [\rho]'$ el conjunto de las transiciones de entrada y $[\rho]_O = \{O_1, ..., O_l\} \subseteq [\rho] \cup [\rho]'$ el conjunto de las transiciones de salida, donde $k + l = m$. Sea $X = \{x_0, ..., x_n\}$ el conjunto de puntos del tiempo global donde $x_i = cl(gt)(s_i)$ $(1 \leq i \leq n)$ es el instante donde $\alpha_i \in [\rho]$ ocurre (tomamos $x_0 = 0$ que es consistente con el hecho de que $x_0 = cl(gt)(s_0)$). Para cualquier transición de salida $a_{O_j} \in [\rho]_O$ con período de habilitación $[\mathsf{start}(O_j), \mathsf{stop}(O_j)]$, $x_{\mathsf{start}(O_j)} \in X$ es el punto en el tiempo donde α_{O_j} pasa a estar habilitada, y $x_{O_j} = x_{\mathsf{start}(O_j)} + cl(\alpha_{O_j})(s_{\mathsf{start}(O_j)})$ es el tiempo en que α_{O_j} se espera que ocurra, donde $cl(\alpha_{O_j})(s_{\mathsf{start}(O_j)})$ es el valor del reloj $cl(\alpha_{O_j})$ en el estado $s_{\mathsf{start}(O_j)}$. Para cada $a'_{O_j} \in [\rho]'_O$, $x_{\mathsf{stop}(O_j)} \in X$ es el tiempo en el que α'_{O_j} pasa a estar deshabilitada.

Para poder calcular la probabilidad de ejecutar un caso de prueba dado, necesitamos establecer *restricciones de tiempo* para el caso de prueba. Primero que todo, los puntos de tiempo global en X están totalmente ordenados, lo cual se deduce de la ejecución secuencial de las transiciones en $[\rho]$.

$$x_0 < x_1 < ... < x_n \tag{5.1}$$

Para cada transición de salida (tomada o habilitada pero no tomada) $\alpha_{O_j} \in [\rho]_O$, el valor inicial del reloj $cl(\alpha_{O_j})$ tiene que ser elegido dentro de las cotas cuando ha sido habilitado.

$$l_{O_j} \leq cl\left(\alpha_{O_j}\right)\left(s_{\mathsf{start}(O_j)}\right) = x_{O_j} - x_{\mathsf{start}(O_j)} \leq u_{O_j} \tag{5.2}$$

Para asegurarse que todas las transiciones de salida $\alpha'_{O_j} \in [\rho]'$ no son ejecutadas, requerimos la siguiente restricción, que pide que cuando α'_{O_j} deja de estar habilitada, estas tienen un valor de reloj mayor a uno,

$$x_{\mathsf{stop}(O_j)} < x_{O_j} \tag{5.3}$$

Notar que si el sistema está compuesto solamente de salidas, entonces todos los x_{O_j} estarían acotados. Agregando entradas puede invalidarse esta propiedad. Como deseamos dar la mejor disposición temporal para alimentar las entradas, podemos de forma segura descuidar sistemas para los cuales este tiempo les es indistinto. Desde ahora, asumiremos que el modelo STIOA y el caso de prueba definen un sistema de desigualdades donde todos los valores de tiempos globales $x_1, ..., x_m$ están acotados.

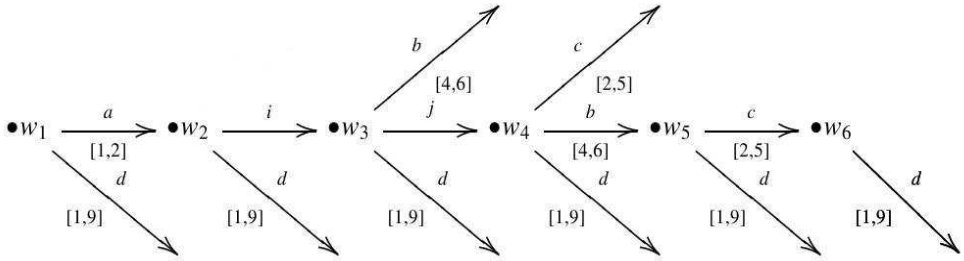
Ejemplo 5.1. Siguiendo el Ejemplo 4.2 y tomando como caso de prueba a $\sigma = aijbc$, podemos ver que tanto b, c como d tienen períodos de habilitación no unitarios. A continuación transcribimos las desigualdades:

$$x_0 < x_a < x_i < x_j < x_b < x_c \quad \text{por} \quad (5.1)$$

$$\begin{array}{rcccl} 1 & \leq & x_a & \leq & 2 \\ 4 & \leq & x_b - x_i & \leq & 6 \\ 2 & \leq & x_c - x_j & \leq & 5 \\ 1 & \leq & x_d & \leq & 9 \end{array} \quad \text{por} \quad (5.2)$$

$$x_c < x_d \quad \text{por} \quad (5.3)$$

obtenidas del sistema que observamos a continuación.



Dada la ejecución ρ, las desigualdades (5.1), (5.2) y (5.3) definen una región B variando sobre dos tipos diferentes de variables: entradas x_{I_i} y salidas x_{O_j}. Las entradas están definidas por el ambiente por eso las consideramos como parámetros externos. Las salidas tienen comportamiento estocástico dado por variables aleatorias independientes en el espacio de probabilidad $(\Omega, \mathcal{F}, P) = ([0,1], \mathcal{B}([0,1]), U([0,1]))$, i. e. el espacio de probabilidad uniforme estándar en el intervalo real $[0,1]$. Cada variable aleatoria es una función lineal $x_{O_j} : [0,1] \to \mathbb{R}^+$ que mapea el intervalo $[0,1]$ para caer en (5.2):

$$x_{O_j}(w) = (u_{O_j} - l_{O_j})w + l_{O_j} + x_{\mathsf{start}(O_j)} \tag{5.4}$$

Entonces definiremos la región paramétrica $B\,(x_{I_1}, ..., x_{I_k})$ como una función que dados los parámetros de entrada nos devuelve la región limitada por variables aleatorias independientes uniformemente distribuidas.

Proposición 5.2. La probabilidad de la *región paramétrica* $B\left(x_{I_1},...,x_{I_k}\right)$ es:

$$P_{x_{O_1},...,x_{O_k}}\left(B\left(x_{I_1},...,x_{I_k}\right)\right)=\frac{V_B\left(x_{I_1},...,x_{I_k}\right)}{\prod_{j=1}^{l}u_{O_j}-l_{O_j}} \tag{5.5}$$

Demostración. Por definición la probabilidad de la región paramétrica $B\left(x_{I_1},...,x_{I_k}\right)$ está dada por la ecuación:

$$P_{x_{O_1},...,x_{O_k}}\left(B\left(x_{I_1},...,x_{I_k}\right)\right)=\int_{B(x_{I_1},...,x_{I_k})}f_1(x_{O_1})...f_l(x_{O_l})dx_{O_1}...\ dx_{O_l}$$

donde f_j es la función de densidad de probabilidad de la variable aleatoria x_{O_j}. Como en este caso $f_j(x)=((x_{\mathsf{start}(O_j)}+u_{O_j})-(x_{\mathsf{start}(O_j)}+l_{O_j}))^{-1}=\frac{1}{u_{O_j}-l_{O_j}}$, tenemos que:

$$P_{x_{O_1},...,x_{O_k}}\left(B\left(x_{I_1},...,x_{I_k}\right)\right)=\int_{B(x_{I_1},...,x_{I_k})}\prod_{j=1}^{l}\frac{1}{u_{O_j}-l_{O_j}}dx_{O_1}...\ dx_{O_l}$$

Despejando las constantes, la probabilidad de una ejecución es:

$$P_{x_{O_1},...,x_{O_k}}\left(B\left(x_{I_1},...,x_{I_k}\right)\right)=\prod_{j=1}^{l}\frac{1}{u_{O_j}-l_{O_j}}\cdot\int_{B\left(x_{I_1},...,x_{I_k}\right)}dx_{O_1}...dx_{O_l}$$

donde la integral es simplemente el volumen de la región paramétrica $B\left(x_{I_1},...,x_{I_k}\right)$ y que es igual al volumen seccional de la región B con k hiperplanos $Vol\left(B\left(x_{I_1},...,x_{I_k}\right)\right)=Vol\left(B\cap H\left(x_{I_1},...,x_{I_k}\right)\right)=V_B\left(x_{I_1},...,x_{I_k}\right)$.

$$P_{x_{O_1},...,x_{O_k}}\left(B\left(x_{I_1},...,x_{I_k}\right)\right)=\frac{V_B\left(x_{I_1},...,x_{I_k}\right)}{\prod_{j=1}^{l}u_{O_j}-l_{O_j}}$$

□

Notar que esta expresión sigue la clásica idea de teoría de probabilidad: la probabilidad es el volumen posible dividido el volumen total.

Las entradas agregan parámetros a la probabilidad por lo tanto es razonable preguntarse cuando esas acciones deben ser realizadas con el objetivo de maximizar la probabilidad del caso de prueba. Para maximizar la probabilidad de ejecución hay que notar la siguiente igualdad:

$$\max_{x_{I_1},...,x_{I_k}}\left(P_{x_{O_1},...,x_{O_k}}\left(B\left(x_{I_1},...,x_{I_k}\right)\right)\right)=\max_{x_{I_1},...,x_{I_k}}\left(\frac{V_B\left(x_{I_1},...,x_{I_k}\right)}{\prod_{j=1}^{l}u_{O_j}-l_{O_j}}\right)=\frac{\max\limits_{x_{I_1},...,x_{I_k}}V_B\left(x_{I_1},...,x_{I_k}\right)}{\prod_{j=1}^{l}u_{O_j}-l_{O_j}}$$

Puesto en otras palabras, para encontrar que $x_{I_1},...,x_{I_k}$ dan la máxima probabilidad de ejecución del caso de prueba debemos encontrar que $x_{I_1},...,x_{I_k}$ dan el máximo volumen de $B\left(x_{I_1},...,x_{I_k}\right)$.

5.2. Cálculo y evaluación de volúmenes paramétricos

Dado un caso de prueba con las transiciones participantes $\alpha_1...\alpha_n\alpha_{n+1}...\alpha_m$ tenemos un sistema generado por las desigualdades (5.1), (5.2) y (5.3) que define la región $B(x_{I_1}, ..., x_{I_k})$. A la hora del cálculo del volumen de esta región si la desigualdad es estricta o no, no afecta el resultado, por lo que tomaremos que todas son $\leq$. Este sistema de desigualdades se puede describir como una desigualdad $A\vec{x} \leq \vec{b}$, donde A es un matriz de tamaño $o \times m$ cuyos elementos elementos pertenecen a $\{-1, 0, 1\}$, o es el número de desigualdades generadas, donde el vector $\vec{x}$ contiene las variables $x_{I_1}, ..., x_{I_k}$ de los tiempos de entrada y $x_{O_1}, ..., x_{O_l}$ de los tiempos de salida y el vector $\vec{b}$ es un vector columna conteniendo constantes enteras de tamaño m. Como la desigualdades del sistema son de la forma $\pm x_i \leq c$ o $x_i - x_j \leq c$ este sistema puede ser representado como una matriz de diferencias acotadas, más conocidas por su sigla en inglés, DBM por "difference bound matrices" [22].

Habiendo establecido a $B(x_{I_1}, ..., x_{I_k})$ como el conjunto de soluciones del sistema $\{\vec{x} \mid A\vec{x} \leq \vec{b}\}$, pasaremos a computar $V(x_{I_1}, ..., x_{I_k})$, que representa el *volumen de un politopo convexo paramétrico* de dimensión l. Para esto, es necesario calcular una integral sobre un poliedro, razón por la cual adaptaremos el método que desarrolló Schechter [36] para, en vez de calcular integrales sobre poliedros, hacerlo sobre politopos convexos paramétricos.

Schechter utiliza el método Fourier-Motzkin para descomponer el poliedro en regiones de integración que ofrezcan límites simples (i.e. afines) para la integración. Un límite de integración es afín si es una combinación lineal de las variables y los coeficientes de las mismas suman 1, o si es una constante. Las siguientes funciones son ejemplos de límites afines $2x{-}y$, $x{-}y + z$, $(x + y + z)/3$ y $\mathbf{k}x + (1 - \mathbf{k})y$.

El método de Fourier-Motzkin [37, Sec 12.2] es el equivalente a la eliminación de Gauss para sistemas de desigualdades, pero a diferencia de este, Fourier-Motzkin agrega nuevas desigualdades al sistema, e incluso es posible que lo haga de manera exponencial.

Este método de eliminación determina cuando un sistema de desigualdades y ecuaciones lineales es consistente o no y, si lo es, nos permite buscar soluciones. Además puede operar con desigualdades estricta y no estrictas, pero en nuestro caso utilizaremos la variante más simple, la que lidia con sistemas $A\vec{x} \leq \vec{b}$.

La eliminación de una variable se realiza realizando operaciones sobre las filas de la matriz aumentada $(A, \vec{b})$, donde A es una matriz $m \times n$. A continuación se realizan los siguientes pasos para eliminar la variable x_1:

1. Reordenar las filas de $(A, \vec{b})$ y multiplicar las filas por constantes positivas como sea necesario de manera tal que la primera columna se una cadena de 1 seguida de una cadena de -1 seguida de 0. En el caso de que una de esas cadenas sea vacía, el sistema es insatisfactible.
   El sistema resultante se observa en la Figura (5.1), donde $r, s, t > 0$, $r + s + t = m$, y tanto $a'_{i',j} = \frac{a_{i,j}}{|a_{i,1}|}$ como $b'_{i'} = \frac{b_i}{|a_{i,1}|}$ valen par de mismo par (i, i') si $a_{i,1} \neq 0$, sino existe un par (i, i') donde $a'_{i',j} = a_{i,j}$ y $b'_{i'} = b_i$.

2. Por cada par $(1, -1)$ de la columna 1 construir una nueva desigualdad sumando las respectivas columnas de $(A, \vec{b})$. La desigualdad resultante es agregada al sistema.
   Al realizar este paso obtenemos el sistema que llamaremos $FM(A, \vec{b})$ y que se observa la Figura (5.2)

$$
\begin{array}{rrcrr}
1 & a'_{1,2} & \dots & a'_{1,n} & b'_1 \\
\dots & \dots & & \dots & \dots \\
1 & a'_{r,2} & \dots & a'_{r,n} & b'_r \\
-1 & a'_{r+1,2} & \dots & a'_{r+1,n} & b'_{r+1} \\
\dots & \dots & & \dots & \dots \\
1 & a'_{r+s,2} & \dots & a'_{r+s,n} & b'_{r+s} \\
0 & a'_{r+s+1,2} & \dots & a'_{r+s+1,n} & b'_{r+s+1} \\
\dots & \dots & & \dots & \dots \\
0 & a'_{r+s+t,2} & \dots & a'_{r+s+t,n} & b'_{r+s+t}
\end{array}
$$

Figura 5.1: Matriz resultante del primer paso de Fourier-Motzkin

$$
\begin{array}{rrcrr}
1 & a'_{1,2} & \dots & a'_{1,n} & b'_1 \\
\dots & \dots & & \dots & \dots \\
1 & a'_{r,2} & \dots & a'_{r,n} & b'_r \\
-1 & a'_{r+1,2} & \dots & a'_{r+1,n} & b'_{r+1} \\
\dots & \dots & & \dots & \dots \\
-1 & a'_{r+s,2} & \dots & a'_{r+s,n} & b'_{r+s} \\
0 & a'_{r+s+1,2} & \dots & a'_{r+s+1,n} & b'_{r+s+1} \\
\dots & \dots & & \dots & \dots \\
0 & a'_{r+s+t,2} & \dots & a'_{r+s+t,n} & b'_{r+s+t} \\
0 & a'_{1,2} + a'_{r+1,2} & \dots & a'_{1,n} + a'_{r+1,n} & b'_1 + b'_{r+1} \\
\dots & \dots & & \dots & \dots \\
0 & a'_{r,2} + a'_{r+s,2} & \dots & a'_{r,n} + a'_{r+s,n} & b'_r + b'_{r+s}
\end{array}
$$

Figura 5.2: Matriz resultante de Fourier-Motzkin

$$M = \left[\begin{array}{cccccc|c} 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 2 \\ -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 9 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 6 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & -4 \\ 0 & 0 & 0 & 1 & -1 & 0 & 5 \\ 0 & 0 & 0 & -1 & 1 & 0 & -2 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 \end{array}\right] \quad M_1 = \left[\begin{array}{cccccc|c} 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 2 \\ -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 9 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 6 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & -4 \\ 0 & 0 & 0 & 1 & -1 & 0 & 5 \\ 0 & 0 & 0 & 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & -1 & 1 & 0 & -2 \\ 0 & 0 & 0 & -1 & 0 & 1 & -4 \\ 0 & 0 & 0 & 0 & 1 & -1 & 6 \\ 0 & 0 & 0 & 0 & 1 & 0 & 7 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 \end{array}\right]$$

Figura 5.3: Matrices aumentadas del Ejemplo (5.1)

Ejemplo 5.3. Las desigualdades correspondientes al Ejemplo (5.1) forman el siguiente sistema $A\vec{x} \leq \vec{b}$:

$$\begin{array}{rrrrrr} +x_a & & & & -x_i & \leq 0 \\ +x_a & & & & & \leq 2 \\ -x_a & & & & & \leq -1 \\ & +x_d & & & & \leq 9 \\ & -x_d & +x_c & & & \leq 0 \\ & -x_d & & & & \leq -1 \\ & & +x_b & -x_c & & \leq 0 \\ & & +x_b & & -x_i & \leq 6 \\ & & -x_b & +x_j & & \leq 0 \\ & & -x_b & & +x_i & \leq -4 \\ & & & +x_c & -x_j & \leq 5 \\ & & & -x_c & +x_j & \leq -2 \\ & & & & -x_j+x_i & \leq 0 \end{array}$$

En la Figura (5.3) vemos al mismo sistema denotado por la matriz aumentada $M = (A \mid \vec{b})$ y la matriz M_1 resultante de aplicar el método de Fourier-Motzkin que también representa al mismo sistema.

El método de Schechter permite calcular integrales sobre un poliedro de n dimensiones,

representado por un sistema de desigualdades P. Para ello descompone el sistema P en un conjunto de sistemas $R = \{R_1, ..., R_m\}$ de n dimensiones y $2n$ desigualdades, de forma tal que

$$\int_P f(x_1, ..., x_n)dx_1...dx_n = \sum_{i=1}^{m} \int_{R_i} f(x_1, ..., x_n)dx_1...dx_n$$

y para R_i los límites de integración para cada variable son afines, tanto por arriba como por abajo.

Para esto, se construye un árbol de $n+1$ niveles, cuyos nodos son sistemas de desigualdades. En el nodo raíz (a nivel 0) se ubica el sistema original. Mantendremos como invariante el hecho de que los nodos de nivel m presentan límites simples para las primeras m-ésimas variables. Entonces para construir los hijos de un nodo de nivel m, tomaremos el sistema resultante de eliminar las primeras $2m$ filas y $2m$ columnas. Sobre este sistema es que aplicaremos Fourier-Motzkin y descompondremos como explicamos en el párrafo siguiente. Cada sistema de la descomposición junto con las $2m$ filas que habíamos descartado forma un nodo de nivel $m + 1$.

La descomposición del sistema $FM(A, \vec{b})$, resultante de aplicar Fourier-Motzkin a la matriz aumentada $(A, \vec{b})$, consiste en $r \times s$ matrices $P_{i,j}$. Cada matriz $P_{i,j}$ con $i \in 1, ..., r$ y $j \in 1, ..., s$ se obtiene de: para $p = 1, ..., r, p \neq i$ reemplazar en $FM(A, \vec{b})$ la fila p por (fila p - fila i) y para $p = r + 1, ..., r + s, p \neq j$ reemplazar en $FM(A, \vec{b})$ la fila p por (fila p - fila $r + j$).

En la Figura (5.4), vemos el sistema resultante respecto al sistema $(A, \vec{b})$ del nodo padre.

Debemos notar que las operaciones de filas realizadas en ambos algoritmos no modifican la posibilidad de caracterizar como DBM a los sistemas resultantes de las descomposiciones.

Ejemplo 5.4. El método Schechter indica que el nodo raíz contiene al sistema M. Para obtener los nodos de nivel 1, tomaremos la resultante de aplicar Fourier-Motzkin a M, en este caso M_1. Observamos que no hay un par de límites afines para la primer variable a eliminar, en este caso x_a, sino una terna de límites lo que no nos permite realizar integral simple.

Para obtener los límites deseados descompondremos la matriz original, creando una matriz nueva por cada par de límites afines posible (cada par conteniendo un límite positivo y otro negativo) que involucre a la variable a eliminar. Además a cada límite no elegido, que incluya la variable a proyectar, le restaremos el límite elegido del mismo signo. El resto de los límites que no involucran a esta variable son mantenidos tal cual. En este ejemplo, donde tenemos 3 límites que incluyen a x_a (2 positivos y uno negativo) crearemos dos nodos de nivel 1, donde en M_1^1 elegimos el segundo límite y se lo restamos al primero y en M_1^2 hacemos lo opuesto. Los nodos resultantes se observan en la Figura (5.5).

Para el siguiente nivel repetiremos el proceso a cada matriz, aplicaremos Fourier-Motzkin y luego descompondremos, pero sin considerar los pares de límites afines obtenidos en el paso anterior. Y así continuaremos hasta obtener un par de límites afines por variable.

Para nuestro problema decidimos modificar ligeramente el algoritmo de [36] de manera tal que al aplicarlo sobre $B(x_{I_1}, ..., x_{I_k})$ las variables que proyectemos sean solamente las salidas $x_{O_1}, ..., x_{O_l}$. Generalizando el método de Schechter para poder indicarle que se detenga al procesar la l-ésima dimensión (o al l-ésimo nivel), obtendremos en el último nivel del árbol

$$
\begin{array}{rrcrr}
1 & a'_{1,2} - a'_{i,2} & \dots & a'_{1,n} - a'_{i,n} & b'_1 - b'_i \\
\dots & \dots & & \dots & \dots \\
1 & a'_{i-1,2} - a'_{i,2} & \dots & a'_{i-1,n} - a'i-_{i,n} & b'_{i-1} - b'_i \\
1 & a'_{i,2} & \dots & a'_{i,n} & b'_i \\
1 & a'_{i+1,2} - a'_{i,2} & \dots & a'_{i+1,n} - a'_{i,n} & b'_{i+1} - b'_i \\
\dots & \dots & & \dots & \dots \\
1 & a'_{r,2} - a'_{i,2} & \dots & a'_{r,n} - a'_{i,n} & b'_r - b'_i \\
-1 & a'_{r+1,2} - a'_{r+j,2} & \dots & a'_{r+1,n} - a'_{r+j,n} & b'_{r+1} - b'_{r+j} \\
\dots & \dots & & \dots & \dots \\
-1 & a'_{r+j-1,2} - a'_{r+j,2} & \dots & a'_{r+j-1,n} - a'_{r+j,n} & b'_{r+j-1} - b'_{r+j} \\
-1 & a'_{r+j,2} & \dots & a'_{r+j,n} & b'_{r+j} \\
-1 & a'_{r+j+1,2} - a'_{r+j,2} & \dots & a'_{r+j+1,n} - a'_{r+j,n} & b'_{r+j+1} - b'_{r+j} \\
\dots & \dots & & \dots & \dots \\
-1 & a'_{r+s,2} - a'_{r+j,2} & \dots & a'_{r+s,n} - a'_{r+j,n} & b'_{r+s} - b'_{r+j} \\
0 & a'_{r+s+1,2} & \dots & a'_{r+s+1,n} & b'_{r+s+1} \\
\dots & \dots & & \dots & \dots \\
0 & a'_{r+s+t,2} & \dots & a'_{r+s+t,n} & b'_{r+s+t} \\
0 & a'_{1,2} + a'_{r+1,2} & \dots & a'_{1,n} + a'_{r+1,n} & b'_1 + b'_{r+1} \\
\dots & \dots & & \dots & \dots \\
0 & a'_{r,2} + a'_{r+s,2} & \dots & a'_{r,n} + a'_{r+s,n} & b'_r + b'_{r+s}
\end{array}
$$

Figura 5.4: Matriz resultante de Schechter

$$M_1^1 = \left[\begin{array}{ccccccc}
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{-2} \\
1 & 0 & 0 & 0 & 0 & 0 & 2 \\
-1 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 1 & 0 & 0 & 0 & 0 & 9 \\
0 & -1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & -1 & 6 \\
0 & 0 & -1 & 0 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 1 & -4 \\
0 & 0 & 0 & 1 & -1 & 0 & 5 \\
0 & 0 & 0 & 1 & 0 & 0 & 9 \\
0 & 0 & 0 & -1 & 1 & 0 & -2 \\
0 & 0 & 0 & -1 & 0 & 1 & -4 \\
0 & 0 & 0 & 0 & 1 & -1 & 6 \\
0 & 0 & 0 & 0 & 1 & 0 & 7 \\
0 & 0 & 0 & 0 & -1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 5 \\
0 & 0 & 0 & 0 & 0 & -1 & -1
\end{array}\right] \quad M_2^1 = \left[\begin{array}{ccccccc}
1 & 0 & 0 & 0 & 0 & -1 & 0 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{2} \\
-1 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 1 & 0 & 0 & 0 & 0 & 9 \\
0 & -1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & -1 & 6 \\
0 & 0 & -1 & 0 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 1 & -4 \\
0 & 0 & 0 & 1 & -1 & 0 & 5 \\
0 & 0 & 0 & 1 & 0 & 0 & 9 \\
0 & 0 & 0 & -1 & 1 & 0 & -2 \\
0 & 0 & 0 & -1 & 0 & 1 & -4 \\
0 & 0 & 0 & 0 & 1 & -1 & 6 \\
0 & 0 & 0 & 0 & 1 & 0 & 7 \\
0 & 0 & 0 & 0 & -1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 5 \\
0 & 0 & 0 & 0 & 0 & -1 & -1
\end{array}\right]$$

Figura 5.5: Nodos de nivel 1 de la descomposición de Schechter para el Ejemplo (5.1)

un conjunto $R_{B(x_{I_1},...,x_{I_k})}$ de subregiones de $B(x_{I_1},...,x_{I_k})$ con límites de integración simples sobre los diferenciales. De cada subregión $R_i^B \in R_{B(x_{I_1},...,x_{I_k})}$, obtendremos de las primeras $2l$ filas los límites superiores e inferiores de la *integral* $Pol(R_i^B)$ correspondiente a cada salida x_{O_i} $(1 \leq i \leq l)$ en función de $x_{I_1},...,x_{I_k},x_{O_1},...,x_{O_{i-1}}$ y de la conjunción del resto de las filas la *condición de aplicación* $AC(R_i^B)$ de esta integral como una función booleana de $x_{I_1},...,x_{I_k}$. Continuar con Schechter no ofrece ninguna ventaja ya que no la hay en tener condiciones de aplicación con pares de limites afines.

La integral $Pol(R_i^B)$ puede ser resuelto por cómputo simbólico recursivamente usando el teorema fundamental del cálculo $\int_a^b x^k dx = \frac{x^{k+1}}{k+1}\Big|_a^b$, obteniendo un polinomio de grado l sobre las k variables de entrada.

La función de evaluación del volumen paramétrico será entonces:

$$V(x_{I_1},...,x_{I_k}) = \sum_{R_i^B \in R_{B(x_{I_1},...,x_{I_k})}} Pol(R_i^B) \times AC(R_i^B)$$

Ejemplo 5.5. En la Figura (5.6), y siguiendo el ejemplo anterior, podemos ver el sistema de un nodo de nivel 4. En este nivel vamos a terminar la descomposición, ya que corresponde a la variable x_c, el tiempo de ocurrencia de la última transición de salida del caso de prueba.

Este nodo corresponde entonces al polinomio

$$Pol(M_4^1) = \int_{4+x_i}^{6+x_i} \int_{4+x_i}^{c} \int_{c}^{9} \int_{1}^{2} dx_a\ dx_d\ dx_b\ dx_c = \frac{22}{3} - 2x_i$$

$$M_4^1 = \left[\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 2 \\ -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 9 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & -4 \\ 0 & 0 & 0 & 1 & 0 & -1 & 6 \\ 0 & 0 & 0 & -1 & 0 & 1 & -4 \\ \hline 0 & 0 & 0 & 0 & 1 & -1 & 4 \\ 0 & 0 & 0 & 0 & 1 & -1 & 2 \\ 0 & 0 & 0 & 0 & 1 & 0 & 7 \\ 0 & 0 & 0 & 0 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 0 & 0 & -1 & -2 \end{array}\right]$$

Figura 5.6: Uno de los nodos resultantes de la descomposición de Schechter para el Ejemplo (5.1)

y a la condición de aplicación

$$AC(M_4^1) = x_i \leq 3 \wedge x_j \leq 7 \wedge -x_i < -2 \wedge x_i - x_j < -1 \wedge x_j - x_i \leq 2$$

Para nuestro ejemplo la función de evaluación completa se puede observar en la Figura (5.7).

Este algoritmo es diferente al presentado en el trabajo original [42]. En el citado trabajo se indica que el conjunto de matrices resultantes de aplicar Schechter nos permite modelar la función de evaluación como una función por partes. Esto no es cierto.

Al aplicar el método original de Schechter las subregiones resultantes de $B(x_{I_1}, ..., x_{I_k})$ no son disjuntas, las intersecciones entre estas sólo se dan en hiperplanos, por lo que [36] indica que para la integración podemos ignorar este hecho.

Pero nuestro problema es diferente, ya que lidiamos con la integral de un politopo convexo paramétrico. Aún aplicando el método original, los hiperplanos resultantes pueden ser disjuntos pero los que forman las condiciones de aplicación no lo serán necesariamente. Por lo tanto la función de evaluación del volumen paramétrico obtenida de este método no es una función por partes.

Este hecho también nos hace preguntar si es conveniente realizar Schechter hasta obtener límites afines para los diferenciales (hasta la dimensión l-ésima) o continuar hasta hacer afines incluso a las condiciones de aplicación. En este trabajo, hemos optado por la primer opción porque es ventajosa respecto al desempeño de la herramienta.

$$
\begin{aligned}
V(x_i, x_j) \quad = \quad & \\
& \left(7 + (-1)\, x_i + 8 x_i x_j + (-1)\, x_i x_j^2 + (-7)\, x_i^2 + 1 x_i^3 + (-8)\, x_j + 1 x_j^2\right) \\
\times \quad & (x_i \leq 2 \wedge x_j \leq 4 \wedge -x_i < -1 \wedge x_j - x_i \leq 4 \wedge x_i - x_j < -1) \\
+ \quad & \left((-9) + 15 x_i + (-7)\, x_i^2 + 1 x_i^3\right) \\
\times \quad & (x_i \leq 2 \wedge -x_i < -1 \wedge x_j - x_i \leq 4 \wedge x_i - x_j < -1 \wedge -x_j < -4) \\
+ \quad & \left((-147) + \frac{245{,}0}{2} x_i + \left(\frac{-119{,}0}{2}\right) x_i x_j + \frac{19{,}0}{2} x_i x_j^2 + \left(\frac{-1{,}0}{2}\right) x_i x_j^3 + \frac{49{,}0}{2} x_i^2 + (-7)\, x_i^2 x_j + \frac{1{,}0}{2} x_i^2 x_j^2 \right. \\
& \quad \left. + \frac{133{,}0}{2} x_j + (-10)\, x_j^2 + \frac{1{,}0}{2} x_j^3\right) \\
\times \quad & (x_i \leq 2 \wedge -x_i < -1 \wedge x_j - x_i \leq 6 \wedge x_j \leq 7 \wedge x_i - x_j < -4) \\
+ \quad & \left(\left(\frac{-22{,}0}{3}\right) + \frac{28{,}0}{3} x_i + (-2)\, x_i^2 + 0 + 0\right) \\
\times \quad & (x_i \leq 2 \wedge -x_i < -1 \wedge x_j - x_i \leq 2 \wedge x_i - x_j < -1) \\
+ \quad & \left(\frac{16{,}0}{3} + \frac{26{,}0}{3} x_i + 7 x_i x_j + (-4)\, x_i x_j^2 + \frac{1{,}0}{3} x_i x_j^3 + \left(\frac{-23{,}0}{2}\right) x_i^2 + 7 x_i^2 x_j + \left(\frac{-1{,}0}{2}\right) x_i^2 x_j^2 + \left(\frac{-8{,}0}{3}\right) x_i^3 \right. \\
& \quad \left. + \frac{1{,}0}{6} x_i^4 + (-14)\, x_j + \frac{9{,}0}{2} x_j^2 + \left(\frac{-1{,}0}{3}\right) x_j^3\right) \\
\times \quad & (x_i \leq 2 \wedge -x_i < -1 \wedge x_j - x_i \leq 4 \wedge x_i - x_j < -2) \\
+ \quad & \left(\left(\frac{-13{,}0}{6}\right) + \frac{20{,}0}{3} x_i + 8 x_i x_j + 1 x_i x_j^2 + \left(\frac{-1{,}0}{3}\right) x_i x_j^3 + (-7)\, x_i^2 + (-4)\, x_i^2 x_j + \frac{1{,}0}{2} x_i^2 x_j^2 + \frac{8{,}0}{3} x_i^3 \right. \\
& \quad \left. + \left(\frac{-1{,}0}{6}\right) x_i^4 + (-4)\, x_j + \left(\frac{-3{,}0}{2}\right) x_j^2 + \frac{1{,}0}{3} x_j^3\right) \\
\times \quad & (x_i \leq 2 \wedge -x_i < -1 \wedge x_j - x_i \leq 1 \wedge x_i - x_j < 0) \\
+ \quad & \left((-7) + (-6)\, x_i + 1 x_i^2 + 8 x_j + (-1)\, x_j^2\right) \\
\times \quad & (x_i \leq 3 \wedge -x_i < -2 \wedge x_j - x_i \leq 4 \wedge x_j \leq 4 \wedge x_i - x_j < -1) \\
+ \quad & \left(9 + (-6)\, x_i + 1 x_i^2\right) \\
\times \quad & (x_i \leq 3 \wedge -x_i < -2 \wedge x_j - x_i \leq 4 \wedge x_j \leq 7 \wedge x_i - x_j < -1 \wedge -x_j < -4) \\
+ \quad & \left(147 + \frac{49{,}0}{2} x_i + (-7)\, x_i x_j + \frac{1{,}0}{2} x_i x_j^2 + \left(\frac{-133{,}0}{2}\right) x_j + 10 x_j^2 + \left(\frac{-1{,}0}{2}\right) x_j^3\right) \\
\times \quad & (x_i \leq 3 \wedge -x_i < -2 \wedge x_j \leq 7 \wedge x_i - x_j < -4) \\
+ \quad & \left(\frac{22{,}0}{3} + (-2)\, x_i + 0 + 0\right) \\
\times \quad & (x_i \leq 3 \wedge -x_i < -2 \wedge x_j - x_i \leq 2 \wedge x_j \leq 7 \wedge x_i - x_j < -1) \\
+ \quad & \left(\left(\frac{-16{,}0}{3}\right) + (-14)\, x_i + 7 x_i x_j + \left(\frac{-1{,}0}{2}\right) x_i x_j^2 + \left(\frac{-5{,}0}{2}\right) x_i^2 + \frac{1{,}0}{6} x_i^3 + 14 x_j + \left(\frac{-9{,}0}{2}\right) x_j^2 + \frac{1{,}0}{3} x_j^3\right) \\
\times \quad & (x_i \leq 3 \wedge -x_i < -2 \wedge x_j - x_i \leq 4 \wedge x_j \leq 7 \wedge x_i - x_j < -2) \\
+ \quad & \left(\frac{13{,}0}{6} + \left(\frac{-9{,}0}{2}\right) x_i + (-4)\, x_i x_j + \frac{1{,}0}{2} x_i x_j^2 + \frac{5{,}0}{2} x_i^2 + \left(\frac{-1{,}0}{6}\right) x_i^3 + 4 x_j + \frac{3{,}0}{2} x_j^2 + \left(\frac{-1{,}0}{3}\right) x_j^3\right) \\
\times \quad & (x_i \leq 5 \wedge -x_i < -2 \wedge x_j - x_i \leq 1 \wedge x_j \leq 4 \wedge x_i - x_j < 0) \\
+ \quad & \left(\frac{125{,}0}{6} + \left(\frac{-25{,}0}{2}\right) x_i + \frac{5{,}0}{2} x_i^2 + \left(\frac{-1{,}0}{6}\right) x_i^3\right) \\
\times \quad & (x_i \leq 5 \wedge -x_i < -3 \wedge x_j - x_i \leq 2 \wedge x_j \leq 7 \wedge x_i - x_j < 0 \wedge -x_j < -4) \\
+ \quad & \left(\frac{49{,}0}{6} + \left(\frac{-49{,}0}{2}\right) x_i + 7 x_i x_j + \left(\frac{-1{,}0}{2}\right) x_i x_j^2 + 14 x_j + \left(\frac{-9{,}0}{2}\right) x_j^2 + \frac{1{,}0}{3} x_j^3\right) \\
\times \quad & (x_i \leq 5 \wedge -x_i < -3 \wedge x_j - x_i \leq 4 \wedge x_j \leq 7 \wedge x_i - x_j < -2 \wedge -x_j < -4)
\end{aligned}
$$

Figura 5.7: Función de evaluación para el Ejemplo (5.1)

5.3. Maximización de probabilidad de casos de prueba

En la ecuación 5.5 la probabilidad de un caso de prueba fue establecida como una fracción con un numerador que es una función de $x_{I_1}, ..., x_{I_k}$, es decir de los tiempos globales en que se realizan las entradas, y un denominador constante independiente del tiempo de las entradas. Entonces el problema de maximizar la probabilidad se reduce al numerador, esto es el volumen del poliedro paramétrico, y dadas las características de la función de evaluación del mismo este se hace computacionalmente tratable por una familia de algoritmos numéricos.

En particular la característica que nos interesa aprovechar es la unimodalidad de la función de evaluación. La prueba de esto radica en el hecho de que representa el volumen seccional de un politopo convexo.

Las desigualdades 5.1, 5.2 y 5.3 forman una intersección de semi-espacios sobre las variables $x_1, ..., x_m$. Siendo estas variables acotadas por la definición del sistema, entonces las desigualdades forman un politopo convexo de dimensión m. La integral $\int_{B(x_{I_1},...,x_{I_k})} dx_{O_1}...dx_{O_l}$es entonces el volumen de $B(x_{I_1}, ..., x_{I_k})$, y esta es una función $V : (\mathbb{R}^+)^k \to \mathbb{R}^+$, donde $x_{I_1}, ..., x_{I_k}$ son los parámetros de entrada del problema de maximización.

Sea $h(x, i)$ el hiperplano en $(\mathbb{R}^+)^m$ definido fijando $x_i = x$ y sea H el subespacio $H(x_{I_1}, ..., x_{I_k}) \doteq \{(x_{I_1}, ..., x_{I_k}, x_{O_1}, ..., x_{O_l}) \mid x_{O_j} \in \mathbb{R}^+, 1 \leq j \leq l\}$ que es intersección de los k hiperplanos $h(x_{I_1}, 1) \cap ... \cap h(x_{I_k}, k)$, cada uno estableciendo un valor de entrada del sistema, sea K el politopo generado por las desigualdades 5.1, 5.2 y 5.3. Entonces la función $V(x_{I_1}, ..., x_{I_k})$ puede también verse como el volumen del politopo convexo $K \cap H(x_{I_1}, ..., x_{I_k})$ que vive en $(\mathbb{R}^+)^m$, por lo tanto la función de evaluación denota el volumen seccional de un politopo convexo [9]. Dado el Corolario 3.13 del Teorema de Brunn-Minkowski, la función de evaluación es unimodal.

En resumen, las características de la función de evaluación que definen que algoritmos es posible utilizar para la maximización son:

- *Unimodalidad:* si el método de maximización converge a un máximo, este será el máximo global. Aun así la función no es cóncava ni incluso continua.

- *No diferenciabilidad:* al estar definida por regiones solapadas, es imposible calcular el gradiente, ya que en los bordes de las regiones es imposible de determinar. Este hecho descarta un gran número de los métodos de maximización.

Ejemplo 5.6. En los gráficos de la Figura (5.8) podemos visualizar la función de evaluación del volumen seccional del Ejemplo 5.5. En la figura de la izquierda vemos la función de volumen y se puede observar una discontinuidad pronunciada en la recta $x_j = x_i$. A la derecha podemos ver como las condiciones de aplicación particionan, en este caso, la región de incertidumbre.

También podemos observar la discontinuidad de la función $V(x_i, x_j)$, fijando $x_i = 2$, y variando x_j. La misma se observa en $x_j = 2$. Y de manera similar, haremos con la no diferenciabilidad y no concavidad, fijando $x_j = 1$, se manifiesta en $x_i = 2$. Sin embargo, en ambos gráficos de la Figura (5.9) se puede notar la propiedad de unimodalidad.

Con esto en mente, decidimos utilizar un método de maximización de funciones multidimensionales que no requiera calcular gradientes. Entre estos métodos se incluye al método de

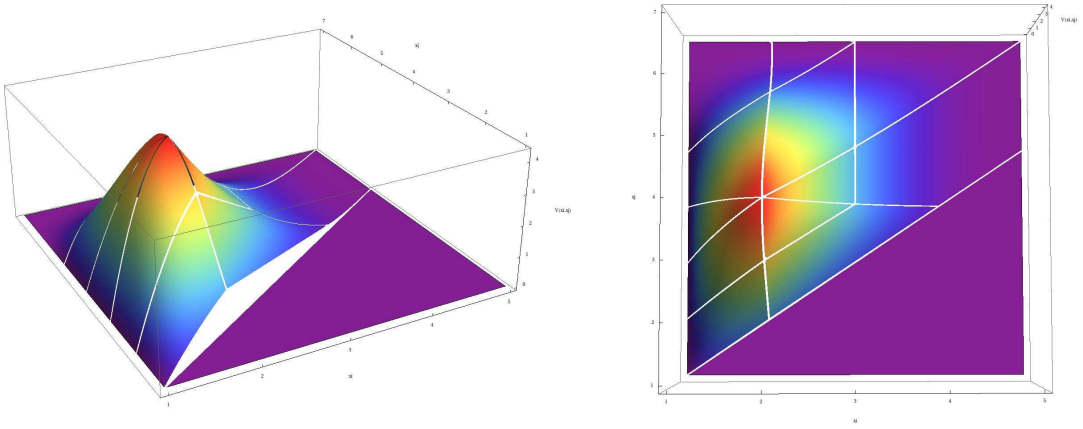


Figura 5.8: Visualización de la función de evaluación del ejemplo (5.5)

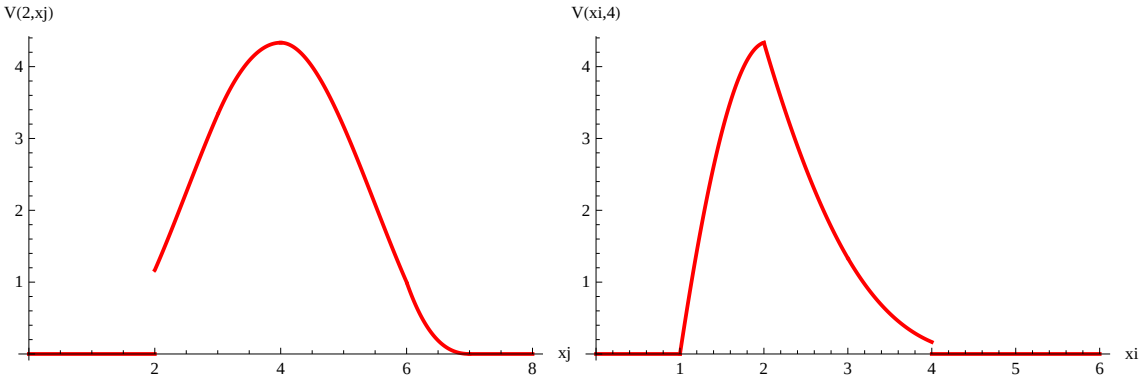


Figura 5.9: Análisis de la función de evaluación del ejemplo (5.5)

colina abajo o Nelder-Mead [35, Sección 10.4] y el de Powell[35, Sección 10.5]. Utilizaremos este último ya que [35] indica que es más rápido en la mayoría de las aplicaciones.

Estos método de maximización, como muchos otros de búsqueda multidimensional, requieren que implementemos búsqueda lineal sobre un vector n-dimensional: dados como entradas los vectores P y n, y la función f encuentra el escalar λ que minimiza $f(P + \lambda n)$, reemplazando P por $P + \lambda n$ y reemplazando n por λn. Implementada la búsqueda lineal, el método multidimensional consiste en elegir la dirección n para la siguiente búsqueda lineal.

La búsqueda lineal que implementaremos es la de Fibonacci, ya que dado un número fijo de evaluaciones esta ha sido probada óptima para encontrar el máximo de una función unimodal [26].

El método de Powell es conocido como uno de los métodos de conjuntos de direcciones. El método más simple de estos, para N dimensiones, consiste en: tomar los vectores unidad $e_1, e_2, ..., e_N$ como conjuntos de direcciones. Usando búsqueda lineal, nos movemos en la primera dirección a su máximo, desde ahí lo hacemos a lo largo de la segunda dirección a su máximo, y así sucesivamente, ciclando por el conjunto de direcciones hasta que la función pare de crecer. Este algoritmo puede ser muy ineficiente para un gran número de funciones, en especial aquellas que tienen valles en algún ángulo respecto de las coordenadas. Como las funciones de volumen seccional tienen esta característica, no decidimos implementar este método que es el citado en [42] ya que no es conveniente.

En cambio, el método de Powell intenta remediar el déficit de este método simple, dando buenas direcciones que nos permitan seguir los valles y que no interfieran entre sí de manera que la maximización en una dirección no sea "pisada" por una dirección siguiente.

El primer paso del método consiste en inicializar el conjunto de direcciones u_i como vectores unitarios:

$$u_i = e_i \qquad i = 1, ..., N$$

Ahora repetiremos la siguiente secuencia de pasos (procedimiento básico) hasta que la función pare de crecer:

- Guardamos la posición inicial como P_0.
- Para $i = 1, ..., N$, movemos P_{i-1} al máximo en la dirección u_i y llamamos a ese punto P_i.
- Para $i = 1, ..., N - 1$, asignamos a $u_i \leftarrow u_{i+1}$.
- Asignamos a $u_N \leftarrow P_N - P_0$.
- Mover P_N al máximo en la dirección u_N y llamar a este punto P_0.

Para garantizar la independencia de las direcciones, vamos a reinicializar el conjunto de direcciones u_i a los vectores unitarios e_i luego de N o $N + 1$ iteraciones de procedimiento básico.

Este método puede conseguir en una pasada de $N(N + 1)$ maximizaciones lineales el máximo para una función cóncava. Es necesario notar que este método no da garantía de convergencia a nuestra maximización, debido a que nuestra función no es cóncava.

Además la búsqueda del máximo se realiza en una región donde en gran parte de la función evalúa a 0 porque no se satisface ninguna condición de aplicación. Debido a ello,

el éxito de la maximización también depende fuertemente del punto en que iniciemos la búsqueda.

Otra asunción crítica para esta clase de algoritmos es que la región de incertidumbre donde se encuentra el máximo debe ser conocida previamente. El siguiente es un procedimiento para obtener el rango inicial de la búsqueda del máximo de la función de evaluación. Recordemos que el sistema definido por 5.1, 5.2 y 5.3 es de la forma de un DBM, por lo que las desigualdades pueden ser de dos formas: $\pm x_i \leq c$ y $x_i - x_j \leq c$. Como pedimos que el sistema sea acotado, lo tomaremos para definir el rango inicial.

En [34], se presenta el algoritmo que denominaremos de Pratt, donde un sistema de desigualdades es codificado como un grafo dirigido con pesos para decidir satisfactibilidad. Los nodos son las variables $x_1, ..., x_m$ con un nodo especial representando el 0. Para cada desigualdad $x_i - x_j \leq c$, el lado $x_i \xrightarrow{c} x_j$ es agregado. En el caso $x_i \leq c$, agregamos el lado $x_i \xrightarrow{c} 0$, cambiando la dirección de la flecha para el caso $-x_i \leq c$. Entonces el camino más corto de x_i a 0 (0 a x_i) es la cota superior mínima (inferior máxima). Usando el algoritmo de Bellman-Ford [10, Secc. 2.3.4] para camino más corto de origen único, en $\mathcal{O}(m^2)$ pasos obtenemos todos los caminos más cortos desde 0 a cualquier x_i, o sea sus cotas inferiores máximas. Dando vuelta la dirección de todos los lados y volviendo a computar obtenemos los caminos más cortos de cualquier x_i a 0, obteniendo las deseadas cotas superiores mínimas.

Parte II

Implementación

Capítulo 6

Descripción del algoritmo

De manera abstracta, vamos a desglosar el algoritmo propuesto por Wolovick, D'Argenio y Qu [42].

Dado un modelo STIOA y un caso de prueba con k entradas y l salidas el algoritmo realiza lo siguiente:

1. Calcular la ejecución de prueba paramétrica con k parámetros (variables de entrada) y l variables ligadas (las de salida), renombrando las entradas y salidas del caso de prueba con el número de ocurrencia de la misma y validando el caso de prueba como un camino del modelo.

2. Derivar el sistema de desigualdades B de $k+l$ variables con las restricciones de tiempos globales de las ocurrencias de las transiciones de prueba. Este sistema de desigualdades tiene las siguientes propiedades:

   a) el conjunto de soluciones es un politopo convexo, o sea un poliedro convexo acotado, de $k+l$ dimensiones,

   b) el mismo está parametrizado en k dimensiones o parámetros,

   c) cada desigualdad del sistema es una diferencia de variables acotada,

   d) las cotas de las desigualdades son números enteros

3. Determinar la satisfactibilidad del sistema de desigualdades B y de esta forma saber si es temporalmente viable la ejecución del caso de prueba provisto. También definir la región de incertidumbre para los k parámetros del sistema o hiperrectángulo que los acota.

4. Calcular el volumen seccional del politopo convexo con k parámetros y l variables ligadas denotado por el sistema de desigualdades B. Lo llamaremos función $V : \mathbb{R}^k \to \mathbb{R}$ y es la integral de la función constante 1 de las l variables ligadas sobre el politopo convexo B.
   Para obtener esta integral, descomponer B de manera tal que cada porción tenga límites de integración simples para cada variable ligada, por ej. que ambos límites sean lineales. Observamos que cada porción a su vez será un politopo convexo de $k + l$ dimensiones con k parámetros.

Formar la función de evaluación, que por cada porción tiene:

   a) la *función de volumen* de la porción: es un polinomio de grado l en k variables. Se puede calcular de manera simbólica usando el teorema fundamental del cálculo y las propiedades de las integrales de funciones polinomiales.

   b) la *condición de aplicación* para la función: es un politopo convexo de k dimensiones.

5. Buscar el vector de $\mathbb{R}^k$, dentro de la región de incertidumbre, que maximiza el volumen seccional del politopo convexo.

6. Calcular la máxima probabilidad de ejecución, dado el volumen máximo y el volumen total de la ejecución de prueba paramétrica.

7. Informar el resultado de la optimización

`TEO` está pensado para ser una implementación de referencia de esta técnica. *"Un lenguaje de programación dinámico y de código abierto enfocado en la simplicidad y productividad [cuya] elegante sintaxis se siente natural al leerla y fácil al escribirla [...]"[3]* parece apropiado para esta tarea. A continuación, discutiremos detalles de esta implementación del algoritmo y las diferencias del algoritmo implementado con el propuesto en [42]. Estos algoritmos serán presentados en el lenguaje de implementación debido a su alta legibilidad y comprensibilidad y con comentarios descriptivos.

6.1. Cálculo de ejecución de prueba paramétrica

Utilizar estrictamente la definición de los STIOA no se consideró como la forma más práctica de modelar nuestro sistema. En su lugar utilizamos una notación de composición paralela de componentes con sincronización de a pares.

Cada componente es un sistema etiquetado de transiciones determinístico y conexo. De cada estado una transición puede tener sólo un estado resultante. Una transición es considerada como salida sólo si se definen las cotas temporales para la ocurrencia de la misma. Por lo tanto, una transición no puede ser de salida y entrada en el mismo componente y todas las ocurrencias de una transición de salida tiene los mismo límites temporales.

La composición paralela de estas componentes determinará un sistema STIOA equivalente. La composición requiere que cada transición de entrada ocurra sólo en un componente. Lo mismo para cada transición de salida. Una transición es considerada como salida de la composición si ocurre como salida de un componente. En el caso que una transición sea salida en un componente y entrada en otro, se la considerará una transición de salida y sincronizará la ejecución de ambos ya que no estará habilitada si no se encuentra habilitada en ambos componentes.

Ejemplo 6.1. El Ejemplo 4.2 ilustra la equivalencia entre esta notación y la definición de STIOA. Gráficamente este es el sistema citado, notar que los símbolos ? y ! se usan sólo para favorecer la comprensión siguiendo la notación habitual de canales de comunicación en notaciones de procesos.

Este sistema se representa en TEO en el siguiente archivo YAML [11]:

```
--- !ruby/object:TEO::Driver::ParallelCompFactory
components:
- - :s1
  - [[:s1, :a, :s2], [:s2, :i, :s3], [:s3, :b, :s4]]
  - [[:b, 4, 6]]
- - :t1
  - [[:t1, :a, :t2], [:t2, :j, :t3], [:t3, :c, :t4]]
  - [[:a, 1, 2], [:c, 2, 5]]
- - :u1
  - [[:u1, :d, :u2]]
  - [[:d, 1, 9]]
test_case: [:a, :i, :j, :b, :c]
```

La primer línea declara la clase con la que se instanciará este objeto. El objeto consta de dos atributos: components y test_case:

- el primero es un arreglo de componentes, donde cada componente es una terna con
  - el estado inicial del componente,
  - un arreglo con las ternas que representan las transiciones (estado inicial, transición y estado final) y
  - otro arreglo de ternas con las transiciones de salida, el limite inferior y el límite superior del intervalo de incertidumbre de ocurrencia de la transición;
- el segundo es un arreglo con las transiciones del camino de prueba.

Una de las suposiciones del algoritmo es que todas las transiciones participantes de una ejecución deben ser diferentes. Por este motivo para calcular la ejecución de prueba paramétrica a partir de un modelo de este tipo y un camino de prueba lo haremos en términos de ocurrencias de transición.

Cada ocurrencia de transición consta de la transición participante, el número de ocurrencia (utilizando 0 para la primera ocurrencia), y el inicio y final del período de habilitación. Una ocurrencia de transición comienza en el momento en que la transición se encuentra habilitada y en el momento anterior o fue ejecutada o no se encontraba habilitada. Esta termina cuando es ejecutada o deja de estar habilitada.

El método comienza con el cálculo de las ocurrencias de transición, luego se agregan las transacciones a la ejecución de pruebas paramétrica, primero las que son ejecutadas en el caso de prueba y luego el resto de las transacciones participantes.

```
#Devuelve una nueva instancia de #ParamTestExecution dado una de
#ParallelComp +model+ y un arreglo con las transacciones de caso de
#prueba +test_case+
def calculate model, test_case
  #Verificar precondición
  check_pre_calculate model, test_case
  #Calcular de ocurrencia de transacciones
  trans_occurrs, test_case_occurrs =
    calculate_trans_occurrs model, test_case
  param_test_exec = TEO::Model::ParamTestExecution.new
  #Agregar ocurrencias de transición ejecutadas
  for trans_occurr in test_case_occurrs
    append_trans param_test_exec, trans_occurr, model, test_case_occurrs
  end
  #Agregar ocurrencias de transición no ejecutadas
  for trans_name, trans_occurr_a in trans_occurrs
    trans_occurr = trans_occurr_a.last
    unless trans_occurr == nil or test_case_occurrs.include? trans_occurr
      append_trans param_test_exec, trans_occurr, model, test_case_occurrs
    end
  end
  #Devolver ejecución de prueba paramétrica
  param_test_exec
end
```

Al calcular las ocurrencias de transición, las ubicaremos en dos contenedores: un arreglo ordenado de la manera que ocurren en el caso de prueba y un hash de arreglos indexados por transición y que contienen sólo las ocurrencias de esta de manera ordenada.

```
#Devuelve una dupla con un hash con todas las ocurrencias de transición
#(#TransOccur), indexadas por nombre y un arreglo de las ocurrencias de las
#transacciones ordenadas como en el caso de prueba
def calculate_trans_occurrs model, test_case
  #Inicializamos contenedores
  trans_occurrs = Hash.new{|hash,key| hash[key]=[]}
  test_case_occurrs = []
  #Pedimos el estado inicial del modelo
  curr_state = model.initial
  #Por cada momento del caso de prueba
  test_case.each_index do |curr_time_point|
    #Pedimos la transición actual
    curr_tc_trans = test_case[curr_time_point]
    #En cada transición del modelo
    model.each_trans do |trans|
      #Obtenemos la ocurrencia de transición actual
      curr_trans_occurrs_a = trans_occurrs[trans]
      curr_trans_occurr = curr_trans_occurrs_a.last
      #Pedimos si la transición está habilitada
      trans_is_enabled = model.enabled?(curr_state, trans)
```

```
      #Si la transición está habilitada
      if trans_is_enabled #
        #Si ya existe una ocurrencia de trans_name,
        #  si esta se encuentra habilitada y,
        #    esta occurrencia no fue ejecutada en el paso anterior
        if curr_trans_occurr and
           (curr_trans_occurr.stop == curr_time_point) and
             (curr_trans_occurr != test_case_occurrs[curr_time_point - 1])
          curr_trans_occurr.continue!
        #es una nueva occurrencia de transición
        #  o si no existe una occurrencia de trans_name
        else
          curr_trans_occurrs_a <<
            TransOccurr.new(
              trans,
              curr_trans_occurrs_a.length,
              curr_time_point,
              curr_time_point + 1)
        end
      # else # si no está habilitada, no hay occurrencia de transición
      end
      #Si es la transición que va a ser ejecutada
      if trans == curr_tc_trans
        #Si no está habilitada, lanzamos una excepción
        raise "The␣transition␣#{trans}␣isn't␣enabled␣at␣#{curr_time_point
}-th␣test␣case␣step" unless trans_is_enabled
        #Asociamos al test_case la occurrencia actual
        test_case_occurrs[curr_time_point] = curr_trans_occurrs_a.last
      end
    end
    #Pedimos el estado siguiente de la ejecución
    curr_state = model.following(curr_state, curr_tc_trans)
  end
  [trans_occurrs, test_case_occurrs]
end
```

6.2. Derivación del politopo convexo paramétrico

Tal como sugiere el trabajo original, representaremos el sistemas de desigualdades derivado de la ejecución de prueba paramétrica como una matriz de diferencias acotadas (DBM) con parámetros. Es decir, elegiremos el dominio abstracto de las DBM paramétricas para representar el politopo convexo paramétrico.

Dicho esto, la derivación es muy directa dadas las desigualdades (5.1), (5.2) y (5.3). En este momento, tendremos en cuenta que el hecho de si la desigualdad es o no estricta no afecta nuestro cómputo, por lo que consideraremos que no lo son. También en este dominio llamaremos a los tiempos de las entradas *parámetros* y a los tiempos de las salidas *variables*, por ser las variables ligadas del sistema.

```
#Devuelve una #ParamDBM asociada a la ParamTestExecution +pte+
def calculate pte
  #Verificamos precondición
  check_pre_calculate pte
  trans = nil
  #Inicializamos contenedores de las dimensiones
  params, vars = [], []
  #Agregamos cada transición tomada en el contenedor respectivo
  pte.each_taken_trans do |trans|
    if pte.is_output_trans?(trans)
      vars << trans
    else
      params << trans
    end
  end
  #Agregamos cada transición de salida no tomada a las variables
  pte.each_enabled_not_taken_trans do |trans|
      vars << trans if pte.is_output_trans?(trans)
  end
  #Inicializamos la DBM paramétrica
  pcp = TEO::Model::ParamDBM.new params, vars
  previous_trans = nil
  #Iteramos cada transición tomada/ejecutada
  pte.each_taken_trans do |trans|
    #Agregamos una ecuación tipo 2
    pcp.add_ineq(previous_trans, trans, 0)
    #Si es de salida
    if pte.is_output_trans?(trans)
      #Agregamos las ecuaciones tipo 3
      pcp.add_ineq(pte.start_trans(trans), trans, -pte.lower_bound(trans))
      pcp.add_ineq(trans, pte.start_trans(trans), pte.upper_bound(trans))
    end
    previous_trans = trans
  end
  #Iteramos por cada transición de salida no tomada/ejecutada
  pte.each_enabled_not_taken_trans do |trans|
    if pte.is_output_trans?(trans)
      #Agregamos las ecuaciones de tipo 3
      pcp.add_ineq(pte.start_trans(trans), trans, -pte.lower_bound(trans))
      pcp.add_ineq(trans, pte.start_trans(trans), pte.upper_bound(trans))
      #Agregamos una ecuación tipo 4
      pcp.add_ineq(pte.stop_trans(trans), trans, 0)
    end
  end
  pcp
end
```

6.3. Determinar la satisfactibilidad del sistema de desigualdades

Determinar la satisfactibilidad del sistema de desigualdades B, nos permite saber si es viable temporalmente la ejecución del caso de prueba provisto. En el mismo paso deberíamos poder calcular también la región de incertidumbre para los k parámetros o hiperrectángulo que los acota, condición útil para implementar el algoritmo de maximización.

Esto se implementa con el siguiente algoritmo.

```
#Devolver la región de incertidumbre para la +param_dbm+ dada
def calculate param_dbm
  #Instanciamos evaluador de satisfactibilidad
  sat_checker = Pratt.new
  #Pasamos las desigualdades al evaluador
  param_dbm.each_ineq do |pos_var, neg_var, bound|
    sat_checker.add_ineq(pos_var, neg_var, bound)
  end
  #Lanzamos una excepción a menos que param_dbm sea satisfactible
  raise "ParamDBM isn't satisfiable" unless sat_checker.is_satisfiable?
  #Instanciamos una región de incertidumbre para los parámetros
  unc_region = TEO::Model::UncRegion.new param_dbm.params
  #Agregamos las cotas de cada parámetro de la región
  for param in param_dbm.params
    unc_region.add_bound(
      param,
      sat_checker.maximal_lower_bound(param),
      sat_checker.minimal_upper_bound(param)
    )
  end
  #Devolvemos la región de incertidumbre
  unc_region
end
```

Como dijimos en la sección 5.3, la representación en DBM del sistema de desigualdades nos permite aplicar el algoritmo propuesto por Pratt [34] y recomendado por el trabajo original.

La implementación se realiza en una clase que crea un grafo dirigido con pesos.

```
class Pratt
  #Constante representando el vértice especial 0
  ZERO = 0

  #Devuelve un verificador de satisfiabilidad de Pratt
  def initialize
    #Inicializamos un grafo dirigido
    @graph = Digraph.new
```

```
    @distance_from_zero = {}; @distance_to_zero = {}
  end

  #add_ineq agrega una desigualdad al verificador representando
  #"pos_var - neg_var >= bound". Tanto pos_var como neg_var
  #puede ser nil, pero no ambos.
  def add_ineq pos_var, neg_var, bound
    raise "Both vars can't be nil" unless pos_var or neg_var
    #Agregamos un lado desde el vértice positivo al vértice negativo
    #con peso bound y usando el vértice ZERO si la var es nil
    @graph.add_edge!(pos_var||ZERO, neg_var||ZERO, bound)
  end
end
```

Usando el algoritmo de Bellman-Ford [10, Secc. 2.3.4] calcularemos las distancias entre las variables y 0, las que dependiendo del sentido de las flechas será la mínima cota superior o la máxima inferior. Este algoritmo requiere que no haya ciclos negativos para obtener las distancias, pero aún en estos casos la implementación devolverá una distancia negativa lo que se informará como no satisfactibilidad ya que representa una contradicción del sistema. Algo similar sucederá con las variables no acotadas cuya distancia será reportada como infinita, en este caso con valor nil (es decir, nulo en Ruby).

```
class Pratt
  #Devuelve true si el sistema con las desigualdades dadas es satisfactible
  def is_satisfiable?
    #Guardamos un hash con la distancia del camino más corto
    #desde ZERO hasta el resto de los vértices usando Bellman-Ford
    @distance_from_zero = @graph.bellman_ford_moore(ZERO)[0]
    #Revertimos el grafo
    rgraph = @graph.reversal
    #Guardamos un hash con la distancia del camino más corto
    #desde el resto de los vértices hasta ZERO usando Bellman-Ford
    @distance_to_zero = rgraph.bellman_ford_moore(ZERO)[0]
    is_satisfiable = true
    #En cada variable/vertice, distinta de ZERO
    for var in @graph.vertices
       max_upper = @distance_from_zero[var]
       opp_min_lower = @distance_to_zero[var]
        #A menos que las distancias desde y hacia ZERO para cada vertice
        #esten definidas, y la cotas superior no sea menor que la menor
        #y la menor no sea que ZERO, el sistema es satisfactible
       unless max_upper and opp_min_lower and
           opp_min_lower >= -max_upper and max_upper <= 0
        is_satisfiable = false
        break
      end
    end
    is_satisfiable
  end
```

```
  #Devuelve la mínima cota superior de var si existe, si no nil
  def minimal_upper_bound var
    @distance_to_zero[var]
  end

  #Devuelve la máxima cota inferior de var si existe, si no nil
  def maximal_lower_bound var
    from_zero = @distance_from_zero[var]
    -from_zero if from_zero
  end
end
```

La implementación concreta de Pratt usa la técnica de memoización para mejorar el desempeño de la misma.

6.4. Calcular el volumen seccional del politopo convexo paramétrico

Para realizar la integral de la función constante 1 de las l variables ligadas sobre el politopo convexo B con k parámetros, el trabajo original sugiere utilizar Schechter y Fourier-Motzkin para descomponer B de manera tal que cada porción tenga límites simples (lineales) de integración para cada variable ligada y condiciones de aplicación simples para cada parámetro. Para luego integrar en cada porción y agregar la función polinomial resultante junto a la respectiva condición de aplicación en la función de evaluación.

En este punto, el trabajo original hace una apreciación errónea al indicar que la función de evaluación es una función por partes. Es cierto que el mismo Schechter [36, pag. 250] indica que las intersecciones entre las porciones obtenidas por la descomposición, si no son vacías, suceden en hiperplanos; que incluso hiperplanos como estos también pueden aparecer en la descomposición; y que ambos hechos pueden ser ignorados en términos de la integración. Estos hechos tienen sentido si integramos en las porciones completas, en definitiva si no tenemos parámetros, pero este no es nuestro caso. Nada indica que las condiciones de aplicación, subpoliedros de k dimensiones de una porción de $k + l$ dimensiones, sean disjuntas, incluso la afirmación anterior nos indicaría que es probable que formen parte de la intersección entre partes.

Dicho esto observamos que la función de evaluación no es una función por partes sino la sumatoria de funciones booleanas (condiciones de aplicación) por funciones polinomiales (integral de variables ligadas). La implementación elegida en este caso para la función de evaluación es un hash donde las funciones polinomiales son acumuladas por condición de aplicación.

El hecho que las condiciones de aplicación no sean disjuntas, elimina la ventaja de proyectar los parámetros del poliedro convexo, por lo que realizaremos la descomposición de Schechter sólo hasta el nivel que garantice que las variables ligadas poseen límites simples de integración. Esto mejora la eficacia de nuestro algoritmo respecto del algoritmo original.

A continuación vemos el algoritmo para calcular el volumen seccional.

```
#Devuelve una #EvalFunction for the given parametrized dbm +param_dbm+
def calculate param_dbm
  #Inicializamos la función de evaluación
  eval_fun = EvalFunction.new param_dbm.params
  #Instanciamos a un árbol de Schechter
  tree = Tree.new(param_dbm)
  #Iteramos por cada nodo en el nivel donde los límites de integración
  #son simples
  tree.each_node_at_level(param_dbm.vars.size) do |subpolytope|
    #Calculamos la condición de aplicación asociada al subpolitopo
    appl_cond = subpolytope.calculate_appl_cond
    #Si la condición de aplicación es satisfactible
    if appl_cond.is_satisfiable?
      #Calculamos la función polinomial con el volumen del subpolitopo
      #realizando la integral de 1 de las variables ligadas
      polinomial = subpolytope.calculate_volume
      #Agregamos la porción en la función de aplicación
      eval_fun.add_piece(appl_cond, polinomial)
    end
  end
  #Devuelve la función de evaluación
  eval_fun
end
```

En las subsecciones siguientes detallaremos este algoritmo.

6.4.1. Descomposición del politopo convexo

Para realizar la integral sobre el poliedro convexo, necesitamos descomponerlo hasta obtener límites simples en las variables a integrar. Schechter propone utilizar el método de eliminación de Fourier-Motzkin sobre una variable del politopo, y del sistema resultante combinar las desigualdades que ligan a esta variable con el hiperplano que forma la proyección de la misma en 0 para obtener una descomposición cuya unión es el politopo original. Con esto en mente, construye un árbol en el que cada nivel representa una nueva variable proyectada.

Antes de comenzar vamos a notar que ambos algoritmos representan el sistema de desigualdades como una *matriz aumentada*, i.e. una matriz donde cada fila representa una desigualdad y cada columna corresponde al coeficiente de la misma variable, excepto la última columna a la derecha que contiene la cota superior de la sumatoria de las variables por sus respectivos coeficientes.

Como queremos proyectar particularmente las variables del politopo convexo paramétrico, ubicaremos a estas en las primeras l columnas del sistema. De esta manera, en el l-ésimo nivel del árbol ya tendremos los subpolitopos con límites simples de integración. Para mejorar aún más el desempeño ordenaremos a las variables de menor a mayor de acuerdo al costo de descomposición en la matriz original. Como el algoritmo proyecta las variables de derecha a izquierda, tomamos como el costo de descomposición al número de subpolitopos que resultarían de descomponer el sistema si fuera la variable a proyectar, i.e. el número de

índices positivos por el de negativos en la columna que representa la variable. Esta idea es sugerida en [39].

Lo descripto en el paso anterior es lo que se realiza al instanciar un árbol de Schechter. En resumen, se provee el politopo paramétrico original, se instancia el nodo raíz, proveyendo los parámetros, las variables y la respectiva matriz aumentada, y por último se ordenan las variables.

```
class Tree
  #Devuelve un Tree dado un politopo paramétrico convexo
  def initialize(system)
    #Instanciamos nodo raíz, obteniendo la matriz aumentada de system
    @system = Node.new(system.params, system.vars, system.to_a, 0)
    #Ordenamos las variables de acuerdo a costo de decomposición
    @system = @system.sort_vars
  end
end
```

Para iterar por los nodos del árbol en el nivel l-ésimo no vamos a crear el árbol hasta el nivel deseado y una vez construido iterar en las hojas del mismo. Esto aumentaría terriblemente el consumo de memoria de nuestro sistema. En nuestra implementación haremos una búsqueda en profundidad l de las hojas de nuestro árbol. La implementación no es la ideal ya que calculamos todos los hijos de cada nodo intermedio en vez de sólo el hijo por el cual vamos a continuar la búsqueda lo que haría al consumo de memoria de a lo sumo l nodos. A pesar de esto el consumo es significativamente menor al que requería el árbol completo.

Entonces el algoritmo que recorre los nodos de cierto nivel es relativamente simple, ya que comienza con una cola cuyo único elemento es el nodo raíz, y continua hasta vaciar la cola, calculando los hijos de un nodo y agregándolos a la cola si no son del nivel deseado, o en caso contrario, pasándolos como argumento al bloque provisto.

```
class Tree
  #Invoca el bloque provisto por cada nodo a nivel level pasandolo
  #como argumento.
  def each_node_at_level level
    #Inicializar cola de procesamiento
    processing = [@system]
    #Mientras la cola no esté vacia
    while not processing.empty? do
      #Tomar el último nodo de la cola
      current_node = processing.pop
      #A menos que el nodo sea del nivel esperado
      unless current_node.level == level
        #Agregar sus hijos a la cola de procesamiento
        processing.concat(current_node.childrens)
      else
        #Ejecutar el bloque provisto con el nodo como parámetro
        yield current_node
      end
```

```
    end
    nil
  end
end
```

Un nodo devuelve sus hijos de acuerdo a Schechter, primero instanciando un sistema apto para Fourier-Motzkin. Recordemos que un nodo de nivel n tiene dos límites por las primeras n variables. Entonces, si no es un nodo del último nivel, partimos el sistema entre las primeras $2n$ filas y las restantes. Sobre estas últimas aplicamos Fourier-Motzkin. A menos que este procedimiento devuelva una excepción, en cuyo caso devolvemos un arreglo vacío, el resultado obtenido es descompuesto como lo describe Schechter y por cada sistema se instancia un nodo del nivel siguiente que es agregado al arreglo resultante. Los sistemas cuyos nuevos límites son iguales, no son agregados al resultado porque forman un hiperplano y por el teorema fundamental del cálculo su integral será 0.

```
class Node
  #Devuelve un arreglo con los hijos del nodo de acuerdo a Schechter
  def childrens
    #Obtener sistema para FM
    system = System.new(@ndim, @rows)
    result = []
    #Si el nivel es menor que la cantidad de variables más parámetros
    if @level < @ndim
      #Calculamos el subsistema no afín
      unaffine_subsys = system.tail @level
      #Nos quedamos con el subsistema afín
      affine_subsys = system.head! @level
      begin
        #Eliminamos la primer variable del sistema no afín
        solution = Solver.solve unaffine_subsys
        #Descomponemos la solución de acuerdo a Schechter
        decomposed_child_syss = solution.decompose @level, affine_subsys
        #Por cada sistema de la descomposición
        for child_sys in decomposed_child_syss
          #Lo agregamos al resultado, a menos que los límites
          #de la variable proyectada sean iguales
          result <<
           Node.new(@params, @vars, child_sys.to_a, @level + 1) unless
              child_sys.are_equal?(@level + 1)
        end
      rescue FourierMotzkin::SystemInfeasible
      #Si el sistema es contradictorio, no tiene hijos
      end
    end
    #Devolvemos el arreglo con los hijos
    result
  end
end
```

El método de eliminación de Fourier-Motzkin es realizado por la biblioteca FM de Louis-Noël Pouchet [33] y se encuentra disponible junto a su documentación en [2].

Como dijimos antes, Schechter indica que en un nodo de nivel n las desigualdades de las filas $2i$ y $2i+1$ nos dicen los límites de la $(i+1)$-ésima variable en función de las $l-i-1$ variables restantes y los k parámetros para $i = 0, ..., n-1$. Tomamos al sistema $\mathcal{A}$ como la matriz con las primeras $2n$ filas y columnas eliminadas (en realidad basta con eliminar las primeras $2n$ filas y eso es lo que implementamos). Habiendo aplicado Fourier-Motzkin a $\mathcal{A}$, el sistema resultante P con las r primeras filas positivas y s primeras negativas será descompuesto de la siguiente manera. Llamaremos P^1 a la proyección de P en la variable n-ésima igual a 0, o sea el sistema P menos las primeras $r+s$ filas. La descomposición constará de los sistemas P^1_{ij} para $i = 0, ..., r-1$ y $j = 0, ..., s-1$ obtenidos con la siguiente receta: para $p = 0, ..., r-1, p \neq i$ reemplazar en P la fila p por (fila p - fila i) y para $p = r, ..., r+s-1, p \neq j$ reemplazar en P la fila p por (fila p - fila $r+j$). La implementación realizada además ubica a las filas i y $r+j$ en las dos primeras filas de P^1_{ij}. Esta descomposición se implementa en los métodos a continuación.

```
class System
  #Devuelve la decomposición de +self+ en la columna +dim_idx+,
  #agregando por delante las filas de +affine_sub_polyhedra+.
  def decompose dim_idx, affine_sub_polyhedra
    #Calcula las listas de índices de filas positivas y negativas en dim_idx
    classify_rows(dim_idx)
    result = []
    #Por cada índice positivo
    for pos_row_idx in @pos_row_idxs
      #Por cada índice negativo
      for neg_row_idx in @neg_row_idxs
        #Agregar al resultado el nodo P^1_ij
        result <<
          FourierMotzkin::System.new(
            @ndim,
            affine_sub_polyhedra.to_a +
              build_p_ij(pos_row_idx,neg_row_idx)
            )
      end
    end
    result
  end

  #Dado el índice de una fila positiva pos_row_idx y una fila negativa
  #neg_row_idx, devuelve una matriz con, el orden dado:
  #* la fila positiva seleccionada,
  #* la fila negativa seleccionada,
  #* las filas con 0,
  #* el resto de las filas positivas menos la fila positiva seleccionada
  #* el resto de las filas negativas menos la fila negativa seleccionada
  def build_p_ij pos_row_idx, neg_row_idx
    result = []
    rest_pos_row_idxs = @pos_row_idxs - [pos_row_idx]
    rest_neg_row_idxs = @neg_row_idxs - [neg_row_idx]
```

```
    p1_rows_idx = ((0...@rows.length).to_a - rest_pos_row_idxs) -
      rest_neg_row_idxs
    for row_idx in p1_rows_idx
      result << @rows[row_idx]
    end
    for row_idx in rest_pos_row_idxs
      offset = -1
      result << @rows[row_idx].collect do |cell_value|
        offset+=1; cell_value - @rows[pos_row_idx][offset]
      end
    end
    for row_idx in rest_neg_row_idxs
      offset = -1
      result << @rows[row_idx].collect do |cell_value|
        offset+=1; cell_value - @rows[neg_row_idx][offset]
      end
    end
    result
  end
end
```

Recordamos que un aspecto muy importante del algoritmo presentado es que todas estas operaciones mantienen los nodos como DBMs. En Fourier-Motzkin, como los sistemas son DBMs los coeficientes ya son 1, 0 ó -1, por lo que *no es necesario normalizar las filas*, y como en el paso siguiente sumamos dos filas cuyo primer coeficiente es 1 y -1, eliminamos una variable y en el caso de que ambas hayan sido diferencias de variables, si no se eliminan entre sí, forman otra diferencia. En la descomposición de Schechter, sucede algo similar, ya que restamos dos filas donde los primeros coeficientes son 1 ó -1.

6.4.2. Determinar condición de aplicación y su satisfactibilidad

En cada porción de la descomposición, es decir en cada nodo en el nivel l-ésimo del árbol, debemos determinar la condición de aplicación de la misma. Como veremos a continuación, en términos de complejidad, el cálculo del volumen de la porción es significativamente mayor incluso que la determinación de la satisfactibilidad de la condición, por lo que sólo haremos este cálculo si la condición de aplicación es satisfactible.

A diferencia del trabajo original, las condiciones de aplicación no poseen sólo $2k$ restricciones o desigualdades, ya que realizamos Schechter sólo a las variables ligadas, pero siguen siendo un politopo convexo de k dimensiones caracterizable como una DBM. Esto hace que la implementación de las condiciones de aplicación sea directa, así como determinar su satisfactibilidad ya que podemos utilizar nuevamente el algoritmo de Pratt.

Determinar la condición de aplicación de una porción consiste en incluir en esta solamente las desigualdades en término de los parámetros. En nuestra implementación, estas son las desigualdades ubicadas a partir de $(2l + 1)$-ésima fila de la matriz aumentada.

Dado que nuestra función de evaluación consiste de un hash indexado por las condiciones de aplicación, debemos implementar una función de hash para las mismas.

6.4.3. Cálculo del volumen de cada porción

Ya hemos dicho que el volumen de cada porción es una función polinomial de k variables. Este se obtiene de la siguiente integral

$$\int_{lower\left(x_{O_l}\right)}^{upper\left(x_{O_l}\right)} \dots \int_{lower\left(x_{O_1}\right)}^{upper\left(x_{O_1}\right)} 1 dx_{O_1} \dots dx_{O_l}$$

donde $upper\left(x_{O_i}\right)$ y $lower\left(x_{O_i}\right)$ son límites lineales de la forma $\pm x_j + c$ con $c \in \mathbb{Q}$ y $j \in \{O_{i+1}, \dots, O_l, I_1, \dots, I_k\}$. Para cada $i = 1, \dots, l$, $upper\left(x_{O_i}\right)$ ($lower\left(x_{O_i}\right)$) se obtiene de la $2i$-ésima ($(2i+1)$-ésima) fila de la matriz subyacente que representa una desigualdad de la forma $x_{O_i} - x_j \leq c$ ($x_j - x_{O_i} \leq -c$) con $j \in \{O_{i+1}, \dots, O_l, I_1, \dots, I_k\}$.

Para realizar el cálculo de la integral utilizamos el siguiente término general

$$\int_{lower(x)}^{upper(x)} c x_1^{k_1} \dots x_n^{k_n} x^k dx = \frac{c}{k+1} x_1^{k_1} \dots x_n^{k_n} \left(upper\left(x\right)^{k+1} - lower\left(x\right)^{k+1} \right)$$

mientras $c, x_1, ..., x_n, x \in \mathbb{R}$ y $k > 0,\ k \in \mathbb{Z}$.

Visto lo anterior y que la integral de un polinomio es igual a la suma de la integral de sus términos, podemos observar que la función del volumen de la porción será una función polinomial de grado l en k variables.

Con esto en mente, implementamos una *representación para polinomios* que contenga su expansión y pueda ser integrada en límites lineales. También nos interesa sumar polinomios, en el caso que en la función de evaluación agreguemos una porción con una condición de aplicación ya existente.

Esta representación utiliza un hash para indexar los términos que lo componen de acuerdo a su parte literal.

```
class Polinomial
  #Devuelve un polinomio
  def initialize
    @terms = {}
  end

  #Agrega un término
  def add_term new_term
    unless new_term.coef == 0
      key = new_term.key
      current_term = @terms[key]
      if current_term then
        current_term << new_term
        @terms.delete(key) if current_term.coef == 0
      else
        @terms[key] = new_term
      end
    end
    self
  end
```

```
  #Actualiza el polinomio por el resultante de la integración
  #de diferencial +diff+ entre los límites (lineales) +upper+ y +lower+
  def integrate! diff, upper, lower
    terms = @terms
    @terms = {}
    for key, term in terms
      term.each_integral_term(diff, upper, lower) do |iterm|
        add_term(iterm)
      end
    end
    self
  end

  #Acumula otro polinomio en este
  def << pol
    for key, term in pol.terms
      self.add_term term
    end
    self
  end
end
```

Como la integración del polinomio es la sumatoria de las integrales de cada término, la implementación de estos debe permitir integrar entre límites lineales, y estos a su vez elevarse a una potencia entera no negativa.

```
class Term
  #Devuelve el término de la forma c * x_1 ** exp_1 * ... * x_n ** exp_n
  #dados:
  #  coef = c
  #  vars = {:x_1 => exp_1, ..., :x_n => exp_n}
  def initialize(coef,vars)
    @coef = coef
    if @coef == 0
      @vars = {}
    else
      @vars = vars
    end
    @key = @vars.to_a
  end

  #Acumular other en self si tienen la mismas vars
  def << other
    @coef += other.coef if @vars == other.vars
  end
```

```
#Invoca el bloque provisto por cada término de la integral definida
  #de self dada la variable de integración diff con los límites upper
  #y lower
  def each_integral_term diff, upper_limit, lower_limit
    if @coef == 0 or upper_limit == lower_limit then
      yield ZERO
    else
      common_factor_vars = @vars.clone
      common_factor_vars.delete(diff)
      integral_degree = (@vars[diff])?(@vars[diff] + 1):(1)
      common_factor =
        Term.new(-@coef.to_f / integral_degree, common_factor_vars)
      lower_limit.each_power_term(integral_degree) do |coef, vars|
        yield(Term.new(coef, vars) * common_factor)
      end
      common_factor.negate
      upper_limit.each_power_term(integral_degree) do |coef, vars|
        yield(Term.new(coef, vars) * common_factor)
      end
    end
    self
  end

  #Actualiza self con el resultado de su multiplicación por obj
  def * other
    new_coef = @coef * other.coef
    if new_coef == 0 then
      return ZERO
    else
      @coef = new_coef
      @vars.merge!(other.vars) do |var,exp,exp_obj|
        exp + exp_obj
      end
      @key = @vars.to_a
      self
    end
  end

  #Transforma el término en su opuesto
  def oppose
    @coef = -@coef
    self
  end
end
```

```
class LinearLimit
  attr_reader :coef, :var

  #Devuelve un límite lineal que representa a coef + var
  def initialize(coef,var)
    @coef = coef
    @var = var
  end

  #Invoca por cada término del resultado de elevar self a la exp-ésima
  #potencia entera no negativa, el bloque provisto con el coeficiente
  #y las variables de cada término como parámetros
  def each_power_term exp
    raise ArgumentError, "Integer >= 0 expected as argument" unless
      exp.kind_of?(Integer) and exp >= 0
    if exp > 0
      yield((@coef ** exp), {}) unless @coef == 0
      if exp > 1 and @var and not (@coef == 0)
        combinat_number = exp.to_f
        for term_number in (1..exp - 1)
          rev_term_number = (exp - term_number)
          yield(combinat_number * (@coef ** rev_term_number),
            {@var => term_number})
          combinat_number =
            combinat_number * rev_term_number / (term_number + 1)
        end
      end
      yield(1, {@var => exp}) if @var
    else #exp == 0
      yield(1, {})
    end
  end
end
```

6.5. Búsqueda del máximo volumen del politopo convexo paramétrico

El siguiente paso del algoritmo requiere que realicemos la búsqueda vector de $\mathbb{R}^k$ que provea el máximo volumen seccional del politopo convexo B.

Como la función de evaluación del volumen es unimodal podemos estar seguros de que la búsqueda del mismo no se detendrá en un máximo local. Pero en una parte significativa de la región de incertidumbre la misma evalúa a 0 e incluso no es continua. Esto quiere decir que nuestra función a maximizar puede no ser cóncava lo que no permite garantizar la convergencia de la búsqueda en el máximo. Notar que cuando nos referimos a no convergencia no nos referimos al hecho de que el algoritmo no termine, sino que la búsqueda termine antes de encontrar el máximo.

Los puntos anteriores no son mencionados en el trabajo original. Incluso el método su-

gerido por el trabajo original para realizar la búsqueda del máximo en un espacio multidimensional puede ser muy ineficiente.

En este trabajo se implementó otro método de búsqueda por conjunto de direcciones, el método de Powell (ver Sección 5.3), que es más eficiente y garantiza convergencia si la función es cóncava.

Otro aspecto a considerar en la búsqueda del máximo es que la evaluación de la función es más costosa de lo previsto. Al no ser una función por partes, no hacemos simplemente búsqueda de la condición de aplicación que aplique para evaluar una función polinomial. En cambio debemos acumular la evaluación de las múltiples funciones polinomiales para las que vale su respectiva condición de aplicación.

6.5.1. Evaluación de la función de volumen

La implementación de la evaluación de la función es muy directa, ya que consiste en acumular el polinomio de cada porción que aplique en los valores dados. Los valores de los parámetros son dados en un hash.

```
class EvalFunction
  def evaluate param_values
    resulting_value = 0
    for appl_cond, polinomial in @pieces
      if appl_cond.evaluate(param_values)
        piece_value = polinomial.evaluate(param_values)
        resulting_value += piece_value unless piece_value < 0
      end
    end
    resulting_value
  end
end
```

La condición de aplicación evalúa cada restricción hasta que encuentra una que devuelva falso, si no lo hace devuelve verdadero. En este sentido, la condición de aplicación es conjunción de las restricciones.

```
class ApplCondition
  def evaluate param_values
    for key,cons in @conss
      return false unless cons.evaluate param_values
    end
    true
  end
end
```

Cada restricción es evaluada realizando la diferencia de los valores de parámetro y devolviendo el resultado de la comparación con la cota. Notar que la evaluación es estricta

cuando la cota es negativa, o sea inferior. Si bien la función de evaluación por definición es unimodal, en la implementación si todos los límites fueran no estrictos, cuando la función fuera evaluada donde se produce la intersección de dos condiciones se obtendría el doble, rompiendo la unimodalidad.

```
class Constraint
  def evaluate param_values
    if @bound > 0
      (param_values[@var]-param_values[@nvar] <= @bound)
    else
      (param_values[@var]-param_values[@nvar] < @bound)
    end
  end
end
```

La evaluación de polinomios y términos es directa

```
class Polinomial
  def evaluate param_values
    result = 0
    for key, term in @terms
      result += term.evaluate(param_values)
    end
    result
  end
end
```

```
class Term
  def evaluate param_values
    result = @coef
    unless @coef == 0
      for var, power in @vars
          result *= (param_values[var] ** power)
      end
    end
    result
  end
end
```

6.5.2. Búsqueda multidimensional por conjuntos de direcciones

Para la búsqueda multidimensional implementamos un método de búsqueda por conjuntos de direcciones, en particular la variante convergente del algoritmo de Powell.

En el trabajo original es propuesta la variante más simple de los métodos de búsqueda por conjuntos de direcciones [29]. Esta consiste, en realizar una búsqueda unidimensional, en

este caso Fibonacci, por cada parámetro de forma recursiva hasta llegar a un punto donde encontramos un valor menor al previamente evaluado. Hay una equivocación en el trabajo original al decir que ese valor se encuentra en tantas búsquedas como parámetros.

El método de Powell luego de haber buscado en tantas direcciones como parámetros y busca en la recta que une el punto con el que empezó y con el que acaba de terminar. Para esto requiere adaptar el método de búsqueda unidimensional.

La terminación de éste método está garantizada debido a que al terminar un ciclo si el valor es menor o igual al actual se detiene la búsqueda. En el caso restante de no terminación sería que nos acerquemos al máximo con un comportamiento de Zenón pero no es posible dada la presición finita del punto flotante.

```
def direction_set_search
  #Calculamos vector base
  basis_vectors = calc_basis_vectors
  set_of_directions = basis_vectors.clone
  #Calculamos el punto de inicio de la búsqueda
  current_point, f_current_point = calc_initial_point
  previous_point, f_previous_point = current_point, f_current_point
  n_iteration = 0
  #Mientras no sean las 2 primeras iteracion y haya nuevo máximo
  while n_iteration < 2 or f_current_point > f_previous_point
    #Guardamos el punto anterior
    previous_point = current_point; f_previous_point = f_current_point
    #Buscamos el máximo en cada dirección del conjunto
    for direction in set_of_directions
      current_point, f_current_point =
        linear_search(current_point, f_current_point, direction)
    end
    #Descartamos la primer dirección del conjunto
    set_of_directions.shift
    #A menos que el punto inicial y el final sea el mismo
    unless current_point == previous_point
      #Agregamos la dirección entre el punto inicial y el último
      set_of_directions <<
        linear_transform(current_point, -1, previous_point)
      #Buscamos el máximo en la nueva dirección
      current_point, f_current_point =
        linear_search(current_point,f_current_point,set_of_directions.last)
    end
    n_iteration += 1
    #Si no hay direcciones o ya cambiamos el conjunto de direcciones
    #completo, reinicializamos el conjunto de direcciones.
    #Esta es la variante convergente de Powell
    if set_of_directions.empty? or
         n_iteration % set_of_directions.size == 0
      set_of_directions = basis_vectors.clone
    end
  end
  [previous_point, f_previous_point]
end
```

Como nuestra función evalúa a 0 para muchos valores y puede no ser cóncava, el punto de comienzo es crítico para converger al máximo en nuestra búsqueda. Para encontrar el punto de inicio vamos a utilizar el hecho que cuando calculamos satisfactibilidad de las condiciones de aplicación, calculamos el mínimo hiperrectángulo que acota a esta. Los *centros de estos hiperrectángulos* son buenos candidatos para encontrar nuestro punto de inicio. En particular elegiremos al que logre la máxima valuación de la función. En caso de que todos estos puntos evalúen a 0, comenzaremos en el centro de la región de incertidumbre.

```
def calc_initial_point
  current_point = nil; f_current_point= 0
  max_point = @unc_region.center
  f_max_point = @eval_fun.evaluate(max_point)
  @eval_fun.each_appl_cond do |appl_cond|
    current_point = appl_cond.center
    if current_point
      f_current_point = @eval_fun.evaluate(current_point)
      if f_current_point > f_max_point
        f_max_point = f_current_point
        max_point = current_point
      end
    end
  end
  [max_point, f_max_point]
end
```

Para la búsqueda lineal, adaptaremos la búsqueda unidimensional de Fibonacci [26], que como el método de la proporción dorada o de Lucas, reducen el intervalo de incertidumbre paso a paso hasta encontrar un intervalo suficientemente reducido para que el error sea menor que la precisión deseada. Esta implementación considera cuidadosamente el caso en que la evaluación de ambos extremos es igual, y utiliza un punto inicial para decidir en cuál de los tres intervalos continuar la búsqueda.

```
#Calcula las constantes de Fibonacci y los
#coeficientes que se utilizan en la búsqueda
def calc_fibonacci_constants
  FIB.replace [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
    1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393,
    196418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887,
    9227465, 14930352, 24157817, 39088169, 63245986, 102334155, 165580141,
    267914296]
  for idx in (0...(FIB.size-2))
    DELTA_FIB[idx] = FIB[-3-idx].to_f / FIB[-1-idx]
  end
end
```

```
#Devuelve false si la búsqueda alcanzó el máximo o la precisión deseada
def continue_search?
  (@_bs_length > $PRECISION_LIMIT and
      @_bs_n_iteration < DELTA_FIB.size) or
    @_bs_n_iteration == 0
end

#Devuelve el nuevo intervalo como indica el método de Fibonacci
#donde @_bs_bottom < delta_bottom < delta_upper < @_bs_upper
def calc_new_bracket
  delta = DELTA_FIB[@_bs_n_iteration] * @_bs_length
  [@_bs_bottom + delta, @_bs_upper - delta]
end

#bracketing_search devuelve el punto máximo y el valor
#respectivo de la función unidimensional
def bracketing_search(init_bottom, init_upper, f_zero)
  @_bs_bottom = init_bottom; @_bs_upper = init_upper
  @_bs_length = @_bs_upper - @_bs_bottom
  @_bs_n_iteration = 0
  max = 0; f_max = Float::MIN # safer

  while continue_search?
    delta_bottom, delta_upper = calc_new_bracket
    f_delta_bottom = yield delta_bottom
    f_delta_upper = yield delta_upper
    if f_delta_bottom < f_delta_upper
      @_bs_bottom = delta_bottom
      (max = delta_upper; f_max = f_delta_upper) if f_delta_upper > f_max
    elsif f_delta_bottom > f_delta_upper
      @_bs_upper = delta_upper
      (max = delta_bottom; f_max = f_delta_bottom) if f_delta_bottom > f_max
    else # f_bo == f_up
      if f_zero > f_delta_bottom # and f_seed > f_up
        if 0 < delta_bottom
          @_bs_upper = delta_bottom
        elsif 0 > delta_upper
          @_bs_bottom = delta_upper
        else
          @_bs_bottom = delta_bottom
          @_bs_upper = delta_upper
        end
      elsif f_zero < f_delta_bottom # and f_seed < f_up
        f_max = f_delta_bottom
        if 0 < delta_bottom
          @_bs_bottom = delta_bottom
          max = delta_upper
        elsif 0 > delta_upper
          @_bs_upper = delta_upper
          max = delta_bottom
        else # seed >= bo or seed <= up
          raise "Situación␣Imposible" unless
               (f_zero - f_delta_bottom).abs < $PRECISION_LIMIT
```

```
        end
      else # f_seed == f_bo == f_up
        break
      end
    end
    @_bs_n_iteration += 1
    @_bs_length = @_bs_upper - @_bs_bottom
  end

  [max,f_max]
end
```

La búsqueda lineal encontrará un escalar a partir del punto en la dirección provista, pero para realizarla debemos dar el intervalo inicial. En nuestro caso será el intervalo que no permita evaluar la función fuera de la región de incertidumbre.

```
#Devuelve tanto el punto como el valor máximo de la función
#de evaluación comenzando en  current_point, que evalua a f_current_point
#,hacia direction
def linear_search init_point, f_init_point, direction
  #Calcular intervalo inicial
  bottom_delta, upper_delta =
    calc_initial_bracketing(init_point, direction)
  scalar=nil
  #Buscar el máximo del bloque siguiente
  max_delta, f_max_point =
    bracketing_search(bottom_delta, upper_delta, f_init_point) do |scalar|
      #Calcular el punto a evaluar
      point = linear_transform(init_point, scalar, direction)
      #Evaluar en nuevo punto
      @eval_fun.evaluate point
  end
  #Calcular el punto máximo con el escalar máximo
  max_point = linear_transform(init_point, max_delta, direction)
  #Devolver punto máximo con valuación máxima
  [max_point, f_max_point]
end

#Devuelve la máxima cota mínima y la mínima cota máxima del escalar
#que genere puntos dentro de la región de incertidumbre
def calc_initial_bracketing(init_point, direction)
  maximal_bottom = nil
  minimal_upper = nil
  #Por cada parámetro de la función de evaluación
  for param in @eval_fun.params
    init_param = init_point[param]
    dir_param = direction[param]
```

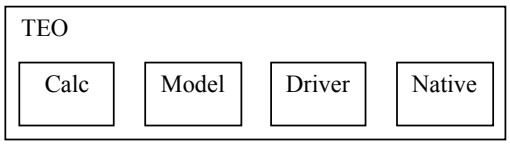
```
    #Calculo las cotas respecto a la dirección en el mismo
     case dir_param <=> 0
     when 0 #if direction[param] == 0
       current_bottom = nil
       current_upper = nil
     when 1 #if direction[param] > 0
       current_bottom =
         (@unc_region.bottom_bounds[param] - init_param).to_f / dir_param
       current_upper =
         (@unc_region.top_bounds[param] - init_param).to_f / dir_param
     else   #if direction[param] < 0
       current_bottom =
         (@unc_region.top_bounds[param] - init_param).to_f / dir_param
       current_upper =
         (@unc_region.bottom_bounds[param] - init_param).to_f / dir_param
     end
     #Evaluo si decido quedarme con las cotas de esta dimensión
     maximal_bottom = current_bottom if (not maximal_bottom) or
        (current_bottom and maximal_bottom < current_bottom)
     minimal_upper = current_upper if (not minimal_upper) or
       (current_upper and minimal_upper > current_upper)
   end
   raise "Null␣bottom␣bound" unless maximal_bottom
   raise "Null␣upper␣bound" unless minimal_upper
   #Devuelvo las cotas para el escalar a buscar
   [maximal_bottom, minimal_upper]
end
```

Capítulo 7

Arquitectura, diseño detallado y metodología de desarrollo de TEO

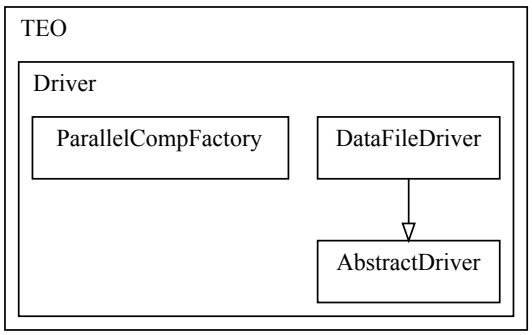
7.1. Arquitectura

La arquitectura de TEO consiste de un módulo para driver (TEO::Driver), uno para los modelos de datos (TEO::Model), otro para la realización de cómputos sobre estos (TEO::Calc), más uno con los mapeos a bibliotecas externas (TEO::Native).

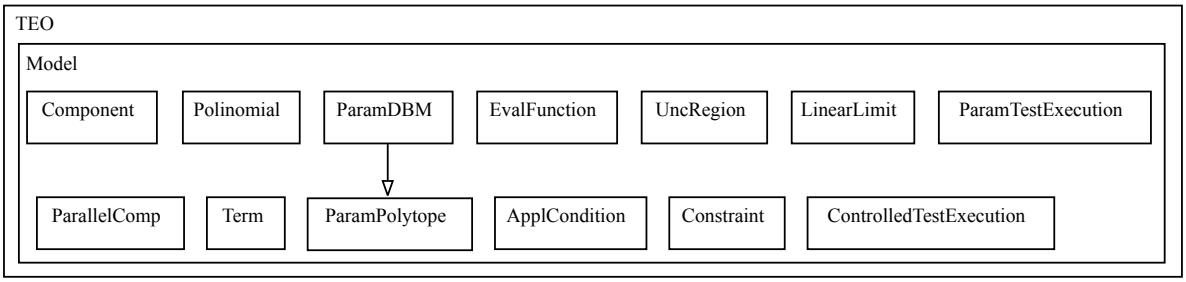


Esta arquitectura realiza una separación de responsabilidades de manera similar a la arquitectura *Model-View-Controller*.

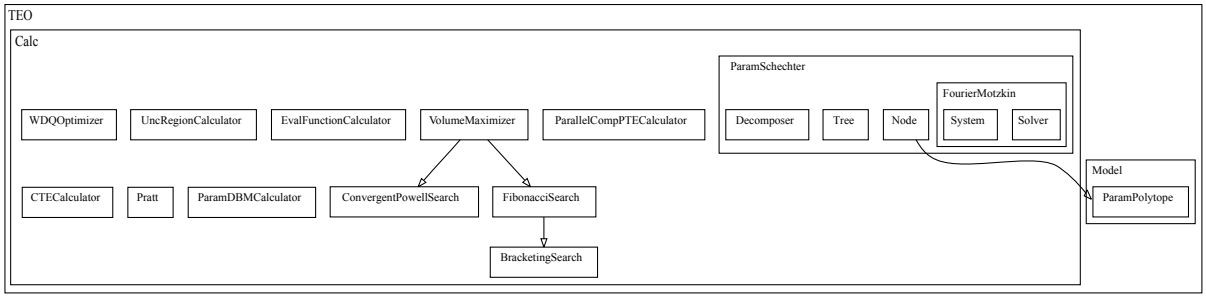
TEO::DRIVER se encarga de cargar los datos necesarios, producir los modelos y pedir los cómputos deseados con estos.



En TEO::Model se ubican los modelos de datos de las entidades operadas en la aplicación.



Las clases de TEO::Calc realizan cómputos y cálculos sobre modelos.



7.2. Diseño detallado

En una ejecución completa de la herramienta mostraremos de qué manera interactúan de manera dinámica los objetos de la implementación.

7.2.1. Driver

En particular, TEO::Driver::DataFileDriver#load lee el modelo STIOA del sistema y del caso de prueba, i. e. una ejecución de prueba, de un archivo YAML con un formato específico. Estrictamente, carga una instancia de TEO::Driver::ParallelCompFactory, que es una fábrica de objetos que devuelve la ejecución de prueba paramétrica inherente al modelo (#obtain_test_execution) y al caso de prueba leído (TEO::Model::ParamTestExecution). Este cómputo se realiza con una instancia de TEO::Calc::ParallelCompPTECalculator. TEO::Driver::DataFileDriver calcula (#calculate) la ejecución controlada de prueba (TEO::Model::ControlledTestExecution) con mayor probabilidad de ejecución, utilizando una instancia de TEO::Calc::WDQOptimizer. Debemos notar que la diferencia entre una ejecución de prueba paramétrica y una controlada consiste en que la segunda indica los tiempos en los cuales realizar las entradas al modelo y la probabilidad de éxito de la ejecución de prueba.

7.2.2. Optimización de la ejecución de prueba

La optimización de la ejecución de prueba (TEO::Calc::WDQOptimizer#optimize) comienza calculando el politopo convexo paramétrico que representa la ejecución de prueba de

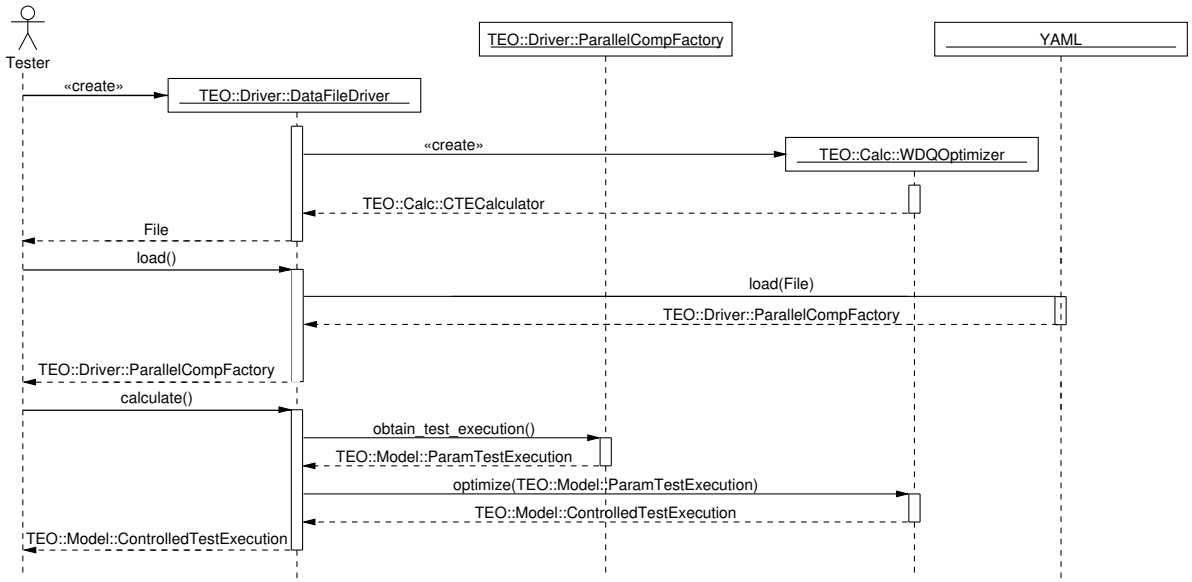


Figura 7.1: Gráfico de secuencia de mensajes esperados por TEO::DRIVER::DATAFILEDRIVER

acuerdo a las desigualdades 5.1, 5.2 y 5.3. En este caso, la representación elegida para este politopo es una matriz de diferencias acotadas paramétrica (TEO::MODEL::PARAMDBM) que calculamos invocando TEO::CALC::PARAMDBMCALCULATOR#calculate. En el paso siguiente, obtenemos la región de incertidumbre (TEO::MODEL::UNCREGION) de los parámetros o hiperrectángulo que los acota invocando el método TEO::CALC::UNCREGIONCALCULATOR#calculate. La optimización sólo podrá realizarse si se verifica que la región de incertidumbre es acotada en cada uno de los parámetros. Luego calculamos la función que define el volumen del politopo convexo paramétrico. La función de evaluación (TEO::MODEL::EVALFUNCTION) es calculada por TEO::CALC::EVALFUNCTIONCALCULATOR#calculate y luego maximizada por TEO::CALC::VOLUMEMAXIMIZER#optimize. Lo que resta es calcular la ejecución de prueba controlada (TEO::MODEL::CONTROLLEDTESTEXECUTION) con los tiempos óptimos para cada entrada y la probabilidad de que se ejecute, invocando TEO::CALC::CTECALCULATOR#calculate.

7.2.3. Satisfactibilidad y región de incertidumbre

El cálculo de la región de incertidumbre (TEO::CALC::UNCREGIONCALCULATOR) es realizado utilizando la implementación del algoritmo de satisfactibilidad de sistemas de diferencias acotadas de Pratt [34] (TEO::CALC::PRATT). Este requiere la implementación de un algoritmo del camino más corto en un grafo dirigido ponderado. En este caso se utiliza el algoritmo de Bellman-Ford-Moore implementado en la biblioteca GRAph Theory in Ruby (GRATR) de acuerdo a [10].

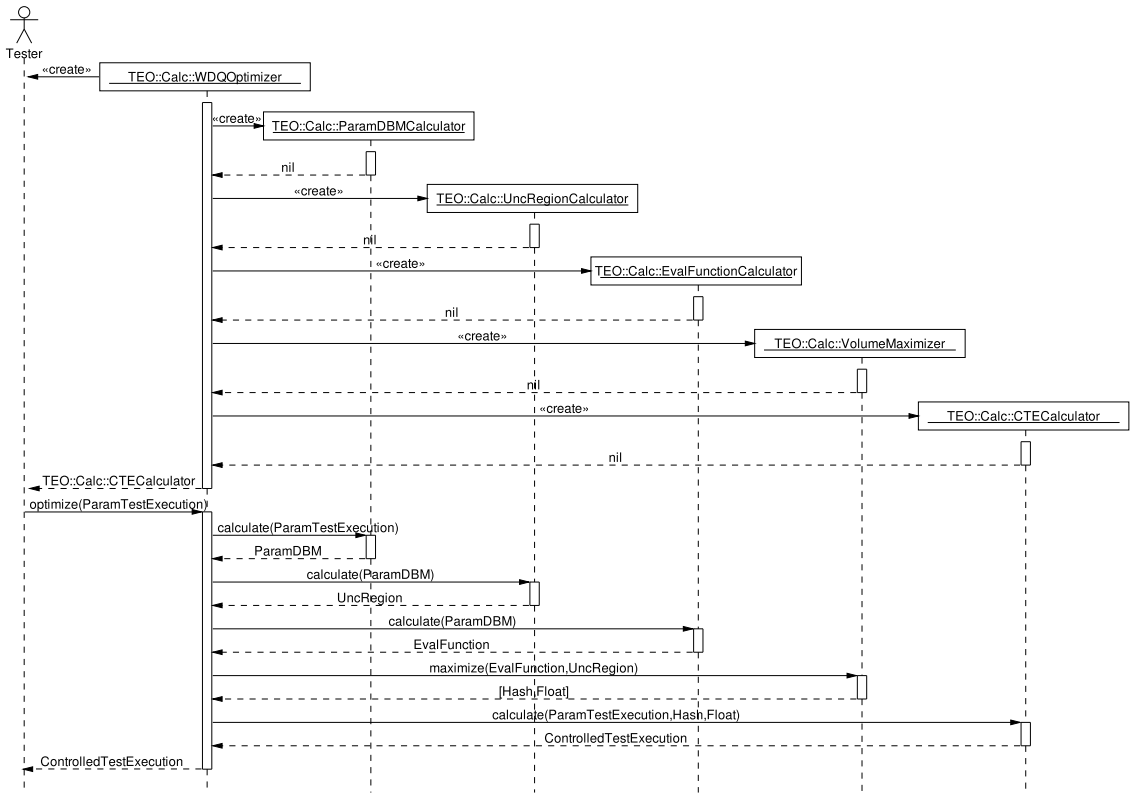


Figura 7.2: Gráfico de secuencia de mensajes esperados por `TEO::Calc::WDQOptimizer`

7.2.4. Cálculo de la función de evaluación

Para el cálculo de la función de evaluación (TEO::CALC::EVALFUNCTIONCALCULATOR) tomamos una DBM paramétrica (TEO::MODEL::PARAMDBM), inicializamos una instancia vacía de la función de evaluación (TEO::MODEL::EVALFUNCTION) y realizamos la descomposición (TEO::CALC::PARAMSCHECHTER::DECOMPOSER#decompose_on_demand) de dicha DBM utilizando el algoritmo de Schechter para la integración sobre poliedros y acumulando el resultado en la función de evaluación. Terminada la descomposición, hacemos inmutable a la función (TEO::MODEL::EVALFUNCTION#inmute) ya que no le agregaremos nuevas porciones durante los cómputos siguientes y de esta manera podremos aplicar optimizaciones para mejorar la velocidad de evaluación de la misma.

7.2.5. Descomposición del politopo paramétrico

Al descomponer bajo demanda la matriz paramétrica de diferencias acotadas, asociada a la ejecución de prueba, (TEO::CALC::PARAMSCHECHTER::DECOMPOSER#decompose_on_demand), se crea una instancia de un árbol de Schechter (TEO::CALC::PARAMSCHECHTER::TREE) y TEO::CALC::PARAMSCHECHTER::TREE#each_node_at_level itera sobre cada subpolitopo convexo paramétrico con límites de integración afines para los diferenciales de las variables que son no parametros. Cada subpolitopo afín es una porción del politopo convexo paramétrico del que calculamos el volumen y sobre cada uno calculamos la condición de aplicación (TEO::MODEL::APPLCONDITION) y la función polinomial (TEO::MODEL::POLINOMIAL) que representa el volumen seccional de esta porción. Ambos objetos son agregados como una nueva porción a la función de evaluación.

7.2.6. Schechter

Cada árbol de Schechter TEO::CALC::PARAMSCHECHTER::TREE al ser instanciado requiere un objeto de una subclase de TEO::MODEL::PARAMPOLYTOPE (en general una instancia de TEO::MODEL::PARAMDBM) y genera el nodo raíz TEO::CALC::PARAMSCHECHTER::NODE del árbol de descomposición. Cuando se solicita iterar sobre los nodos de un determinado nivel (TREE#each_node_at_level) se piden los hijos de cada nodo (NODE#childrens) hasta encontrar los del nivel indicado y se invoca el bloque provisto al método con el nodo como argumento por cada uno.

Un nodo en el árbol de Schechter (TEO::CALC::PARAMSCHECHTER::NODE) denota un politopo convexo paramétrico, y la unión de los nodos del mismo árbol de cada nivel denotan al mismo politopo convexo paramétrico, en particular al que se encuentra en la raíz. Dado el dominio abstracto utilizado en el algoritmo de Schechter, cada nodo está representado como una matriz aumentada paramétrica. Cuando se solicitan los hijos de un nodo (NODE#childrens) se crea un sistema (TEO::CALC::PARAMSCHECHTER::FOURIERMOTZKIN::SYSTEM) equivalente al que denota el nodo y al cual se le aplicará el método de eliminación de Fourier-Motzkin (TEO::CALC::PARAMSCHECHTER::FOURIERMOTZKIN::SOLVER#solve). El sistema resultante se descompondrá de acuerdo a Schechter (SYSTEM#decompose) y de este podemos crear los nodos hijos. De cada nodo con limites afines de integración podremos extraer la condición de aplicación (NODE#calculate_appl_cond) y la función polinomial de volumen del mismo (NODE#calculate_volume).

7.2.7. Fourier-Motzkin

Para implementar TEO::Calc::ParamSchechter::FourierMotzkin::Solver#solve utilizamos en TEO::Native::FM la biblioteca DL para ligarnos con la biblioteca nativa "FM: the Fourier-Motzkin library" [2] de Louis-Noël Pouchet. En realidad se utiliza una versión modificada para que no aborte y no se produzcan fugas de memoria en el caso que el sistema sea contradictorio. Esta biblioteca es nativa y debe ser compilada con GCC 4.3.

7.2.8. Maximización de la función de evaluación

Dadas la función de evaluación (TEO::Model::EvalFunction) y la región de incertidumbre de los parámetros de la misma TEO::Model::UncRegion, buscamos los valores de estos parámetros que nos den la valuación máxima de esta función. TEO::Calc::VolumeMaximizer realiza esta tarea utilizando el método convergente de Powell para búsqueda acotada del máximo en funciones multidimensionales (TEO::Calc::ConvergentPowellSearch) y Fibonacci como método de búsqueda acotada en funciones lineales (TEO::Calc::FibonacciSearch). En particular VolumeMaximizer liga estos algoritmos genéricos con el problema en que trabajamos, en particular con la representación de un vector: la transformación lineal de un vector, la evaluación de la función a maximizar y los cálculos de las cotas para la búsqueda multi- y uni-dimensional. También se define la elección del punto de partida y el vector de direcciones básicas, que utilizando el conocimiento de dominio pueden mejorar la convergencia y eficiencia de la búsqueda.

7.3. Detalles de la implementación y metodología de desarrollo

Como ya comentamos el objetivo de este trabajo es construir una implementación de referencia de esta técnica para optimizar la ejecución de casos de prueba. Decidimos trabajar con software libre porque de esta manera se garantiza que esta sea accesible a quienes estén interesados en hacer otra implementación. Para esto basta con que el ambiente de ejecución sea libre, pero es deseable que las herramientas de desarrollo también lo sean. A continuación describiremos cómo fue el proceso de desarrollo y qué técnicas y herramientas se utilizaron en el proceso.

El desarrollo se realizó de manera incremental e iterativa con la ejecución continua de pruebas tanto unitarias como de integración. No era requisito del trabajo implementar todos los algoritmos utilizados por la técnica por lo que se evaluaron implementaciones de terceros.

En esta etapa se eligió como lenguaje de desarrollo a Ruby [3, 24]. Además de las características ya mencionadas soporta múltiples plataformas, e incluso permite trabajar con bibliotecas nativas, lo que es sumamente interesante si planeamos utilizar herramientas de desarrollo.

La primer etapa de desarrollo se concentró en los dos pasos más críticos para el desarrollo del algoritmo: calcular la función de evaluación y obtener el máximo volumen paramétrico. Esta etapa comenzó con una implementación parcial del problema. Durante el desarrollo del trabajo original se realizaron pruebas de concepto del algoritmo en cuadernos de Mathematica [6]. Estos cuadernos realizaban ambos pasos utilizando las características de alto nivel

de la herramienta. Nuestro interés era aprovecharlos no como implementación de referencia, sino como oráculo para nuestras pruebas.

Para estos pasos del algoritmo, evaluamos implementaciones de Schechter, Fourier-Motzkin y computación simbólica. No evaluamos implementaciones de matrices de diferencia acotada (DBM) ya que tanto Schechter como Fourier-Motzkin están definidos en término de matrices aumentadas y se encuentra disponible al menos una implementación optimizada de Fourier-Motzkin [2], no así de Schechter. Si bien se encontraron sistemas de álgebra computacional (CAS por sus siglas en inglés) libres, como Maxima [4], Axiom [1], Yacas [7], estos no ofrecen interfaces de programación en su distribución, sino servidores e interfaces gráficas o de línea de comandos que interpretan el lenguaje propio de cada sistema. Dada la complejidad de la implementación (establecer un mecanismo de comunicación de baja latencia con el servidor, además de parsear y generar expresiones en un lenguaje especializado) y lo específico del problema a resolver, i.e. integrar polinomios, decidimos desarrollar este cómputo por nuestros medios.

A continuación, se definió el algoritmo de alto nivel que consistía en estos dos pasos. Si bien siguen estando implementados en Mathematica, en esta etapa se establecieron los modelos de datos tanto para las entradas (politopo convexo paramétrico, región de incertidumbre), estructuras intermedias (función de evaluación del volumen) y salidas (vector de parámetros y volumen correspondiente). Una vez implementados los modelos de datos y las pruebas de estos pasos, desarrollamos los módulos que proveerán la misma funcionalidad, pero implementando la técnica indicada.

El paso siguiente a implementar fue la obtención de la región de incertidumbre desde un politopo convexo paramétrico, en concreto, el algoritmo de Pratt. En este caso encontramos una biblioteca nativa de Ruby para trabajar con grafos.

En las siguientes etapas, se implementan la carga de STIOAs desde un formato de archivo específico, la obtención de la ejecución paramétrica y el politopo convexo paramétrico correspondiente.

Una vez que la herramienta ya fue funcional, comenzamos el proceso de optimización y refactorización del código. Fue esencial para la optimización poseer herramientas para medir el uso de recursos de los diferentes bloques funcionales del sistema, y para la refactorización tener un conjunto de prueba de regresión exhaustivo que nos provea confianza de que los cambios de diseño no afectan el desempeño y, sobre todo, la funcionalidad.

Parte III

Resultados

Capítulo 8

Experimentos

8.1. Pruebas con diferentes ejecuciones

Tanto [27] como [42] nos ofrecen ejemplos para probar nuestras técnicas. Utilizaremos estos sistemas y variantes más complejas para ilustrar el comportamiento de la herramienta.

La primera ejecución de prueba no posee entradas y es la planteada por Jurdzinksky et ál. en [27]. Para este caso no habrá maximización ni función de evaluación y el único resultado que obtendremos es la probabilidad de la ejecución. Ver Figura 8.1.

La ejecución de ejemplo de un STIOA de [42], utiliza un sistema con tres componentes que compiten entre sí, mientras dos de ellos largan de manera sincronizada. Ver Figura 8.2.

Una variante al sistema anterior, es un sistema con cuatro componentes, con una de ellas sincronizando a otras dos con el siguiente caso de prueba. Ver Figura 8.3.

Otra variante es replicar las componentes sincronizadas y el caso de prueba, duplicando las condiciones de carrera. Ver Figura 8.4.

Por último, construimos una variante *cíclica* del sistema de *"example v"* donde ejecutaremos dos veces el mismo caso de prueba. Ver Figura 8.5.

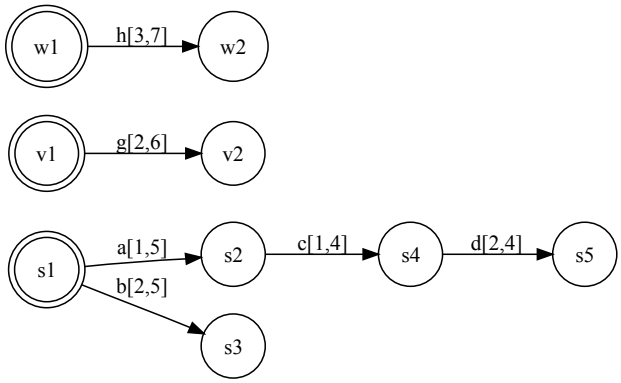


Figura 8.1: Este sistema y el caso de prueba $\delta = ag$ forman la ejecución de prueba *"jurdzinski"*

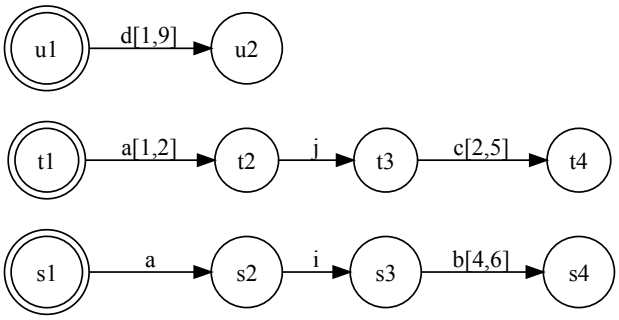


Figura 8.2: Este sistema y el caso de prueba $\delta = a\boldsymbol{ij}bc$ forman la ejecución de prueba *"example V"*

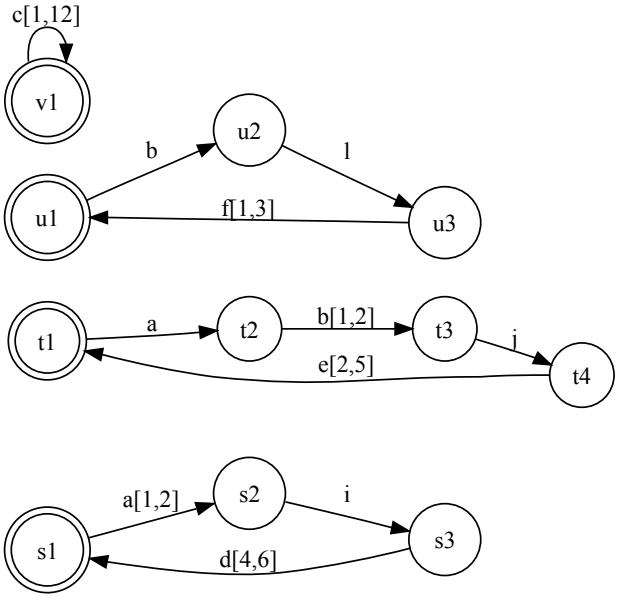


Figura 8.3: Este sistema y el caso de prueba $\delta = ab\boldsymbol{ijl}def$ forman la ejecución de prueba *"triple"*

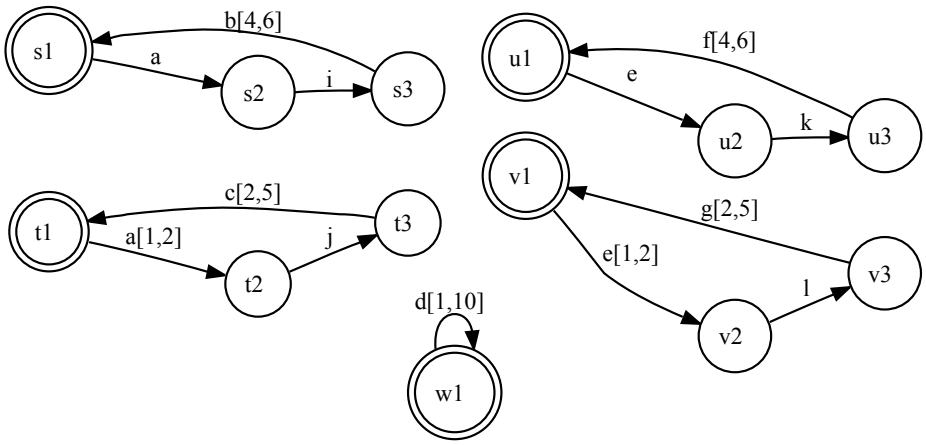


Figura 8.4: Este sistema y el caso de prueba $\delta = ae\boldsymbol{ikjl}bfcg$ forman la ejecución de prueba *"double"*.

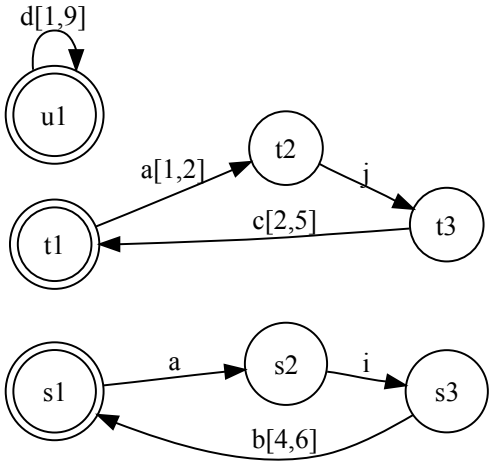


Figura 8.5: Este sistema y el caso de prueba $\delta = a\boldsymbol{ij}bcda\boldsymbol{ij}bc$ forman la ejecución de prueba *"twice"*

8.2. Resultados obtenidos

`TEO` obtuvo los siguientes parámetros óptimos con sus respectivas probabilidades de ejecución en los casos estudiados, ordenados por complejidad en su cálculo.

Ejecución	k	l	Probabilidad	Parámetros
jurdzinski	0	2	0,25850694	$\{\ \}$
example v	2	3	0,09027777	$\{i_0 = 2; j_0 = 4\}$
triple	3	5	0,0533899	$\{i_0 = 3,5815; j_0 = 5,5815; l_0 = 7,5815\}$
double	4	6	0,00956489	$\{i_0 = 2; k_0 = 2,2961; j_0 = 4,5302; l_0 = 4,9088\}$
twice	4	7	0,00766653	$\{i_0 = 2; j_0 = 3,9695; i_1 = 9,7137; j_1 = 11,9183\}$

Cuadro 8.1: Probabilidad de ejecución de los casos estudiados con `TEO`

El sistema de álgebra computacional *Mathematica* sólo pudo resolver los dos primeros casos obteniendo resultados idénticos a `TEO`. En el resto de los casos no devolvió ningún resultado incluso durante ejecuciones de ocho o más horas.

Ejecución	k	l	Probabilidad	Parámetros
jurdzinski	0	2	$\frac{1489}{5760}$	$\{\ \}$
example v	2	3	0.0902778	$\{i_0 = 2; j_0 = 3{,}99999\}$
triple	3	5	*N/A*	*N/A*
double	4	6	*N/A*	*N/A*
twice	4	7	*N/A*	*N/A*

Cuadro 8.2: Probabilidad de ejecución de los casos estudiados con *Mathematica*

8.3. Análisis de desempeño

Para nuestras pruebas utilizamos un sistema *Intel Core2 Quad Q9400* de 2,67 Ghz con *4GB 800MHz DDR2* corriendo *openSUSE 11.2 (x86_64)* con kernel *linux 2.6.31.*

Los resultados siguientes se obtuvieron con el intérprete Ruby de mejor desempeño en el sistema de prueba, *Ruby Enterprise Edition 1.8.7 2009.10*[5]. Además este intérprete provee soporte para medir consumo de memoria.

Los resultados a continuación representan los promedios de 30 ejecuciones consecutivas de cada caso.

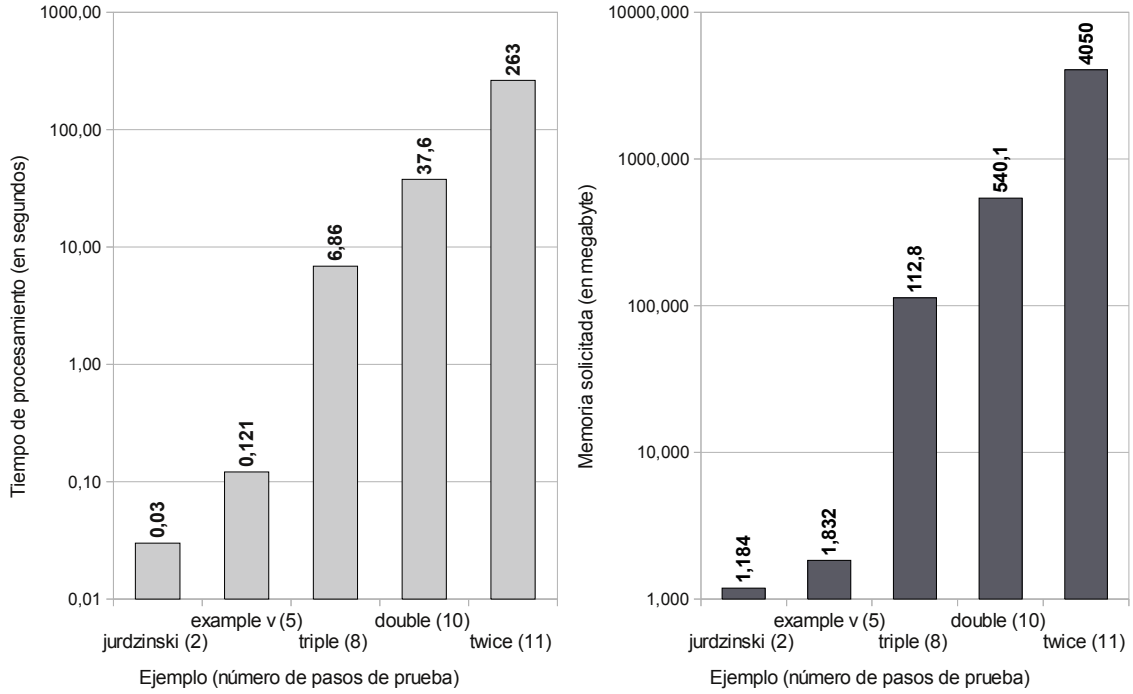


Figura 8.6: Desempeño de los casos estudiados

Vemos que la elección de las ejecuciones no fue azarosa. Cada escenario intenta representar ejecuciones donde no es trivial decidir la competencia entre tiempos de entrada y tiempo de salida para obtener la máxima probabilidad de ejecución. Y de escenario a escenario tratamos de hacer cada vez más compleja la competencia introduciendo más pasos en los casos de prueba.

Dicho esto y analizando los tiempos de procesamiento y la cantidad de memoria solicitada por cada ejecución podemos deducir que, al menos para los peores casos, hay un crecimiento exponencial de los recursos consumidos ante el aumento de pasos en el caso de prueba. También se observa cierta correlación entre el aumento del tiempo de procesamiento y el consumo de memoria.

Teniendo en cuenta que hay un aumento de la complejidad en cada caso estudiado, es interesante también analizar cómo se distribuyen los recursos en cada paso del algoritmo.

En la Figura 8.7 podemos observar que, salvo el caso no parametrizado, la maximización del volumen seccional generalmente utiliza la mitad del tiempo de ejecución, y que el resto del tiempo es para calcular la función de evaluación, donde los pasos significativos son la descomposición de Schechter incluyendo Fourier-Motzkin, el cálculo de las condiciones de aplicación y su satisfactibilidad, y el cálculo de la función polinomial.

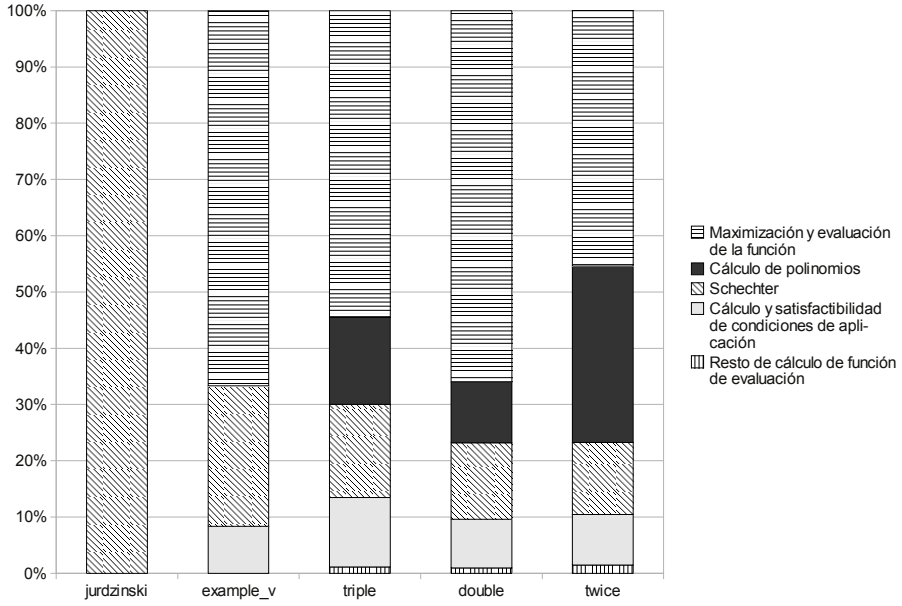


Figura 8.7: Distribución de tiempo de procesamiento en cada ejemplo

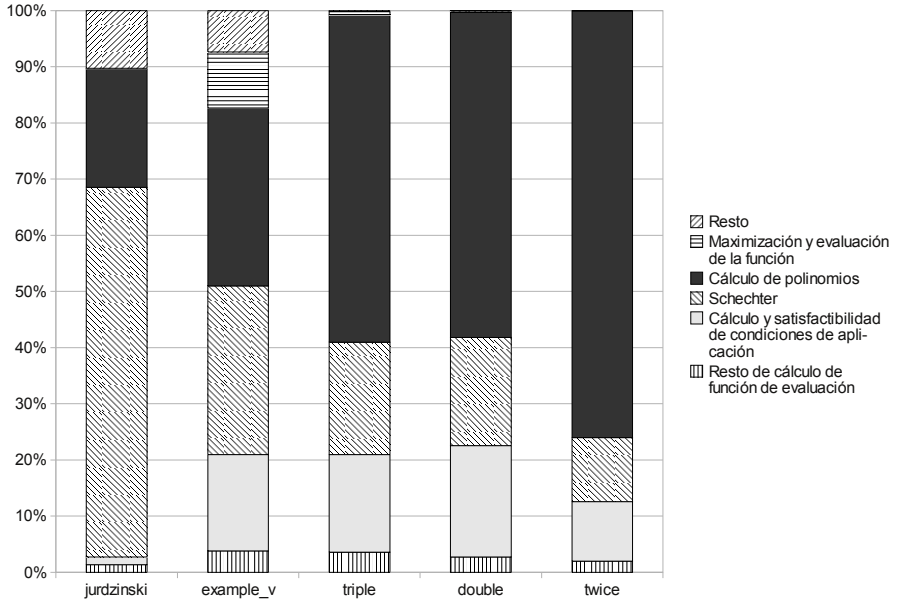


Figura 8.8: Distribución de memoria solicitada en cada ejemplo

En el caso del consumo de memoria (Figura 8.8), hay dos puntos muy interesantes a destacar:

- como la maximización consiste fundamentalmente en múltiples evaluaciones de la función de volumen, prácticamente no solicita memoria[1];
- el cálculo de las funciones polinomiales que representan el volumen de cada porción consume mayor cantidad de memoria a medida que el caso es más complejo.

[1]La maximización consume memoria, en su mayoría hash de diferentes puntos, por eso en *example_v* todavía es una carga (10 %) pero la cantidad de puntos es prácticamente lineal al número de parámetros, por lo que a medida que aumenta la complejidad y crece el consumo global de memoria exponencialmente, este consumo es cada vez menos significativo. En *jurdzinski*, no hay maximización.

Capítulo 9

Conclusiones

La técnica para optimizar la ejecución de casos de prueba para autómatas de entrada/salida estocásticos temporizados de [42] posee ahora una implementación de referencia. Esta también funciona para el problema en sistemas no controlados (sin entradas) como el expuesto en [27].

Durante el desarrollo también se encontraron defectos en [42], particularmente el hecho de la función de evaluación resultante no es una función por partes sino una sumatoria condicionada de funciones polinomiales y la maximización de la misma no es convergente con el método de direcciones simples. Ambos hechos aumentan la complejidad del problema, justamente porque al no ser cóncava la función de evaluación no se puede garantizar convergencia y las evaluaciones son más costosas por no sólo ser una búsqueda de la condición de aplicación, sino la acumulación de todos los polinomios que cumplan.

La herramienta permite completar la generación de los casos de prueba, no sólo indicando los tiempos apropiados para las entradas sino también la probabilidad de ejecución de los mismos. Estos casos de prueba no sólo pueden ser ejecutados sino también utilizados para validar el modelo. Incluso si no estamos interesados en la probabilidad de la ejecución de un caso de prueba, podemos validar esta traza al menos para saber si es posible de ejecutar, lo cual es muy útil para validar un algoritmo de generación de casos de prueba para este sistema.

Como trabajo final de la licenciatura, este trabajo es interesante pues permite experimentar el enfoque de las ciencias de la computación en su campo de trabajo. Un problema de ingeniería de software, como ciertamente es el testing de sistemas de tiempo real, es abstraído a la rigurosidad matemática de un modelo computacional. Ya en este dominio se plantea la equivalencia entre el problema original, la optimización de ejecuciones de prueba, y uno geométrico, la maximización del volumen seccional de un politopo convexo. Ahora, este es desglosado en un problema de álgebra computacional (Fourier-Motzkin y Schechter), otro de grafos (Bellman-Ford usado por Pratt), uno de computación simbólica (integración simbólica para obtener funciones polinomiales), e incluso uno de cálculo numérico (la maximización de un función multivariable), entre otros. Con el algoritmo en mente, volvemos a tener otro problema de ingeniería de software: cómo implementar la herramienta. Ahora diseñamos una arquitectura del programa, definimos estructuras de datos, evaluamos el proceso y el ambiente de desarrollo, para finalmente programar no sólo el programa en sí, sino las mismas pruebas que evidenciarán su funcionamiento. Y al final, pero no por ello menos importante, presentar un informe con los fundamentos y pruebas de su funcionamiento. Este

trabajo representa el enfoque y propósito de nuestra formación para desempeñarnos ya sea como profesionales o como académicos.

9.1. Trabajos futuros

Los dos aspectos centrales a mejorar de esta herramienta son: el cálculo de la función de evaluación y la optimización de la misma.

En el primer área, el cálculo de las funciones polinomiales es el aspecto principal a estudiar. Ya sea a través de la utilización de una interfaz con un sistema de álgebra computacional externa (Yacas, Axiom o Maxima) o la implementación de un algoritmo más sofisticado de integración y manipulación de polinomios [19]. Otro aspecto a mejorar es el hecho de transformar matrices de diferencias acotadas a una representación más costosa como matrices aumentadas para poder realizar Fourier-Motzkin y Schechter. Se podrían adaptar ambos algoritmos a este dominio abstracto, o utilizar las variantes de Fourier-Motzkin en el dominio de desigualdades de dos variables (conocidas como TVPI, del inglés two variables per inequation) de complejidad polinomial [8] o $n^{\log n}$ [31] y adaptar Schechter a este dominio. Como último aspecto de este problema, evaluar si hay alternativas para integrar sobre un poliedro [14].

Respecto del problema de la maximización hay dos aspectos susceptibles de mejora: la evaluación de la función y el método de maximización. En esta implementación, la evaluación consiste en decidir por cada una de las condiciones de aplicación si es verdadera, y en ese caso, acumular el resultado de evaluar la función polinomial. Entonces la complejidad de la evaluación es proporcional al largo de la función. Una alternativa para acotar este recorrido sería reducir esta búsqueda a las condiciones de aplicación cuyos hiperrectángulos que acotan contengan el punto a evaluar. Estructuras como árboles X [12] o similares se podrían utilizar para indexar y buscar en estas cajas. Entre los métodos de optimización, es conocido que Brent en [18] ofrece las variantes óptimas para búsqueda de máximo en funciones no derivables. Estas son significativamente más complejas de implementar que los métodos seleccionados, lo que no era recomendable tal vez para una implementación de referencia.

Otro aspecto a estudiar es la posibilidad de paralelizar la implementación. Es sabido que las operaciones sobre matrices, los métodos de maximización e incluso cómputos simbólicos pueden aprovechar este cambio de paradigma. Además ya se han estudiado las variantes de algunos algoritmos utilizados en esta técnica, como por ejemplo Fourier-Motzkin en [28] y Powell en [21].

Por último, y tal vez la extensión más obvia, sea agregar en la herramienta la funcionalidad que permita realizar optimizaciones "al vuelo" como se indica en la última sección del trabajo original.

Bibliografía

[1] Axiom Computer Algebra System. Disponible en: http://axiom-developer.org.

[2] FM: the Fourier-Motzkin library. Disponible en: http://www-roc.inria.fr/~pouchet/software/fm.

[3] Lenguaje de Programación Ruby. Disponible en: http://www.ruby-lang.org/es.

[4] Maxima CAS. Disponible en: http://maxima-project.org.

[5] Ruby Enterprise Edition. Disponible en: http://www.rubyenterpriseedition.com.

[6] Wolfram Mathematica: Technical Computing Software. Disponible en: http://www.wolfram.com/products/mathematica.

[7] The Yacas computer algebra system. Disponible en: http://yacas.sourceforge.net.

[8] Ash, R. B., y Doléans-Dade, C. A. *Probability & Measure Theory, Second Edition*, 2 ed. Academic Press, 1999. Disponible en: http://books.google.com/books?isbn=9780120652020.

[9] Avis, D., Bose, P., Toussaint, G., Shermer, T. C., Zhu, B., y Snoeyink, J. On the sectional area of convex polytopes. En *Proceedings of the twelfth annual symposium on Computational geometry* (Philadelphia, Pennsylvania, United States, 1996), ACM, págs. 411–412. Disponible en: http://dx.doi.org/10.1145/237218.237411.

[10] Bang-Jensen, J., y Gutin, G. Z. *Digraphs: Theory, Algorithms and Applications*. Springer, 2008. Disponible en: http://books.google.com/books?isbn=9781852332686.

[11] Ben-Kiki, O., Evans, C., y Ingerson, B. YAML Ain't Markup Language (YAML) (tm) Version 1.0. Inf. téc., YAML.org, September 2004. Disponible en: http://www.yaml.org/spec/1.0.

[12] Berchtold, S., Keim, D. A., y Kriegel, H.-P. The X-tree: An Index Structure for High-Dimensional Data. En *Proceedings of the 22th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1996), VLDB '96, Morgan Kaufmann Publishers Inc., págs. 28–39. Disponible en: http://portal.acm.org/citation.cfm?id=645922.673502.

[13] Berkenkötter, K., y Kirner, R. Real-Time and Hybrid Systems Testing. En *Model-Based Testing of Reactive Systems*. Springer, 2005, págs. 355–387. Disponible en: http://dx.doi.org/10.1007/11498490_16.

[14] Bernardini, F. Integration of polynomials over n-dimensional polyhedra. *Computer-Aided Design 23*, 1 (1991), 51 – 58. Disponible en: http://dx.doi.org/10.1016/0010-4485(91)90081-7.

[15] Bourque, P., y Dupuis, R. Guide to the Software Engineering Body of Knowledge 2004 Version. *Guide to the Software Engineering Body of Knowledge 2004 SWEBOK*, 1 (2004), i–D–7. Disponible en: http://books.google.com/books?isbn=0769523307.

[16] Brandán Briones, L., y Brinksma, E. A test generation framework for quiescent real-time systems. *Formal Approaches to Software Testing 3395* (2005), 2004. Disponible en: http://dx.doi.org/10.1007/978-3-540-31848-4_5.

[17] Brandán Briones, L., y Röhl, M. Test derivation from timed automata. *Lecture notes in computer science 3472* (2004), 201–231. Disponible en: http://dx.doi.org/10.1007/11498490_10.

[18] Brent, R. *Algorithms for Minimization Without Derivatives*. Dover books on mathematics. Dover Publications, 2002. Disponible en: http://books.google.com/books?isbn=0486419983.

[19] Bronstein, M. *Symbolic integration I: transcendental functions*. Algorithms and computation in mathematics. Springer, 2005. Disponible en: http://books.google.com/books?isbn=3540214933.

[20] Clarke, D., y Lee, I. Automatic test generation for the analysis of a real-time system: Case study. En *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE* (Montreal, Quebec , Canada, Jun. 1997), págs. 112–124. Disponible en: http://dx.doi.org/10.1109/RTTAS.1997.601349.

[21] Dennis, J. E., y Torczon, V. Direct Search Methods on Parallel Machines. *SIAM Journal on Optimization 1*, 4 (1991), 448–474. Disponible en: http://dx.doi.org/10.1137/0801027.

[22] Dill, D. Timing assumptions and verification of finite-state concurrent systems. En *Automatic Verification Methods for Finite State Systems*, J. Sifakis, Ed., vol. 407 de *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1990, págs. 197–212. Disponible en: http://dx.doi.org/10.1007/3-540-52148-8_17.

[23] Fecko, M., Uyar, M., Duale, A., y Amer, P. A technique to generate feasible tests for communications systems with multiple timers. *Networking, IEEE/ACM Transactions on 11* (2003), 796–809. Disponible en: http://dx.doi.org/10.1109/TNET.2003.818182.

[24] Flanagan, D., y Matsumoto, Y. *The Ruby programming language*. O'Reilly Series. O'Reilly, 2008. Disponible en: http://books.google.com/books?isbn=0596516177.

[25] Gardner, R. J. The Brunn-Minkowski Inequality. *American Mathematical Society 39* (2002), 355–405. Disponible en: http://dx.doi.org/10.1090/S0273-0979-02-00941-2.

[26] Johnson, S. M. Best exploration for maximum is Fibonaccian. Inf. téc., Rand Corp, 1956. Disponible en: http://www.rand.org/pubs/research_memoranda/RM1590.

[27] Jurdziński, M., Peled, D., y Qu, H. Calculating Probabilities of Real-Time Test Cases. En *Formal Approaches to Software Testing*, W. Grieskamp and C. Weise, Eds., vol. 3997 de *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, págs. 134–151. Disponible en: http://dx.doi.org/10.1007/11759744_10.

[28] Kessler, C. W. Parallel Fourier-Motzkin Elimination. En *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II* (London, UK, 1996), Euro-Par '96, Springer-Verlag, págs. 66–71. Disponible en: http://dx.doi.org/10.1007/BFb0024686.

[29] Krolak, P., y Cooper, L. An extension of Fibonaccian search to several variables. *Communications ACM 6* (1963), 639–641. Disponible en: http://dx.doi.org/10.1145/367651.367694.

[30] Luo, G., von Bochmann, G., y Petrenko, A. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *Software Engineering, IEEE Transactions on 20* (1994), 149–162. Disponible en: http://dx.doi.org/10.1109/32.265636.

[31] Nelson, C. G. An n^{logn} algorithm for the two-variable-per-constraint linear programming satisfiability problem. Inf. téc., Stanford University, Stanford, CA, USA, 1978. Disponible en: ftp://reports.stanford.edu/www/TR/CS-TR-78-689.html.

[32] Nielsen, B., y Skou, A. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer (STTT) 5* (Nov. 2003), 59–77. Disponible en: http://dx.doi.org/10.1007/s10009-002-0094-1.

[33] Pouchet, L.-N., Bastoul, C., Cohen, A., y Cavazos, J. Iterative optimization in the polyhedral model: Part II, multidimensional time. En *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)* (Tucson, Arizona, June 2008), ACM Press, págs. 90–100. Disponible en: http://dx.doi.org/10.1145/1375581.1375594.

[34] Pratt, V. Two easy theories whose combination is hard, Sep. 1977. Disponible en: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.112.8263.

[35] Press, W. H. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992. Disponible en: http://dx.doi.org/10.2277/0521431085.

[36] Schechter, M. Integration over a Polyhedron: An Application of the Fourier-Motzkin Elimination Method. *The American Mathematical Monthly 105* (Mar. 1998), 246–251. Disponible en: http://www.jstor.org/stable/2589079.

[37] Schrijver, A. *Theory of Linear and Integer Programming*. John Wiley & Sons, Chichester, 1986. Disponible en: http://books.google.com/books?isbn=0471982326.

[38] Schütz, W. Fundamental issues in testing distributed real-time systems. *Real-Time Systems 7*, 2 (1994), 129–157. Disponible en: http://dx.doi.org/10.1007/BF01088802.

[39] Simon, A., y King, A. Exploiting Sparsity in Polyhedral Analysis. En *Static Analysis*, C. Hankin and I. Siveroni, Eds., vol. 3672 de *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, págs. 336–351. Disponible en: http://dx.doi.org/10.1007/11547662_23.

[40] Springintveld, J., Vaandrager, F., y D'Argenio, P. R. Testing timed automata. *Theoretical Computer Science 254* (Mar. 2001), 225–257. Disponible en: http://dx.doi.org/10.1016/S0304-3975(99)00134-6.

[41] Tretmans, J. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems 29* (Dic. 1996), 49–79. Disponible en: http://dx.doi.org/10.1016/S0169-7552(96)00017-7.

[42] Wolovick, N., D'Argenio, P. R., y Qu, H. Optimizing Probabilities of Real-Time Test Case Execution. En *ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation* (Washington, DC, USA, 2009), IEEE Computer Society, págs. 446–455. Disponible en: http://dx.doi.org/10.1109/ICST.2009.59.