Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Optimización Sistemática de Simulaciones a Eventos Discretos para Sistemas Complejos de Gran Escala: Contribuciones al Simulador PowerDEVS

Tesis de Licenciatura en Ciencias de la Computación

Gerónimo Romczyk

Director: Rodrigo Castro

Codirector: Ezequiel Pecker Marcosig

Buenos Aires, 2025

Agradecimientos

Antes que nada le quería agradecer a Rodri por haberme llamado para las becas BIICC e iniciarme en esto que tanto me gusta, y por seguir apoyándome en los proyectos por venir. Eternamente agradecido.

Muchas gracias también a Eze por codirigir esta tesis y, además de trabajar incansablemente codo a codo, ser un excelente compañero.

Muchas gracias a Mati por sus inmensos aportes.

Muchas gracias a mi familia por bancarme en todas y hacer que esto sea posible.

Muchas gracias a mis amigos, a mi novia y nuevamente a mi familia por hacer que todo valga la pena.

Y muchas gracias al Departamento de Computación, a Exactas y a la Universidad de Buenos Aires por brindarme una educación de excelentísima calidad.

Espero poder devolver todo lo que me han dado y más.

Abstract

Esta tesis presenta un conjunto integral de contribuciones orientadas a optimizar el rendimiento computacional y la flexibilidad de PowerDEVS, un motor de simulación ampliamente utilizado para sistemas de eventos discretos. Entre los aportes principales se encuentra la optimización del núcleo del motor de simulación, generando reducciones significativas en los tiempos de ejecución, junto con el desarrollo de nuevas herramientas de profiling que permiten detectar y comparar automáticamente regresiones y mejoras de performance entre distintas versiones del software. Además, se implementó un sistema de logging configurable y eficiente, que posibilita un trazado detallado cuando se encuentra habilitado, y en caso contrario tiene un impacto mínimo en los tiempos de simulación.

Se introdujo el aplanamiento automático de modelos DEVS jerárquicos, simplificando estructuras internas y reduciendo la complejidad de la mensajería. Asimismo, se optimizó el manejo de conexiones entre modelos mediante el rediseño de estructuras de datos, acelerando la propagación de eventos. Se modernizó el sistema de scheduling de eventos, incorporando algoritmos alternativos de colas de prioridad (como Pairing Heap y Splay Tree), logrando mejoras de velocidad en diversos escenarios de benchmark.

A nivel de modelos atómicos, se adaptó el integrador QSS3, empleando funciones numéricas más precisas y eficientes para el cálculo de raíces cúbicas. Los DataLoggers fueron mejorados mediante estructuras de datos optimizadas, acelerando el cálculo de métricas estadísticas. Finalmente, los modelos de redes de datos fueron adaptados para minimizar la sobrecarga de registro de variables, facilitando simulaciones a gran escala.

Los casos de estudio abarcan modelos sintéticos (DEVStone), modelos reales

de una red de comunicaciones (para la adquisición de datos del experimento ATLAS en el CERN) y modelos epidemiológicos (MultiSIR), demostrando aceleraciones relativas de hasta un 66 % en simulaciones a gran escala.

En conjunto, estos avances consolidan a PowerDEVS como una plataforma eficiente para experimentación numérica en ciencias e ingeniería, ampliando sus posibilidades para simulaciones a gran escala y en tiempo real y sientan las bases para el análisis sistemático de futuras mejoras.

Índice general

Índice de figuras

Parte I

Introducción, Objetivos y Conceptos Preliminares

1

Introducción

1.1 Contexto general y motivación

En las últimas décadas, se ha observado un aumento sostenido en la complejidad de los sistemas estudiados en diversas disciplinas científicas y tecnológicas. Este crecimiento de la complejidad, unido a la necesidad de reducir los tiempos y costos asociados al desarrollo y análisis de sistemas, ha impulsado el surgimiento de técnicas *eficientes* de diseño, evaluación y validación. En este contexto, la simulación basada en modelos se ha consolidado como una herramienta fundamental para la experimentación numérica, especialmente cuando las técnicas analíticas tradicionales o los métodos heurísticos no son viables debido a limitaciones prácticas o computacionales.

La simulación permite analizar el comportamiento dinámico de sistemas complejos antes de su implementación física, facilitando la identificación temprana de problemas, la evaluación de alternativas de diseño y la optimización de su desempeño. Sin embargo, a medida que los modelos aumentan en tamaño, detalle y complejidad, la *eficiencia de la simulación* puede convertirse en un obstáculo crítico. En muchos casos, una baja performance en la simulación puede tornar inviable todo un proyecto basado en experimentación numérica, socavando así las ventajas iniciales que motivaron su elección.

1.2 Desafíos actuales en la optimización del desempeño de simulaciones

Ante este panorama, la optimización del rendimiento de simulaciones computacionales se ha convertido en un área clave dentro del ámbito del modelado y la simulación. Numerosos enfoques han sido desarrollados con el objetivo de mejorar el desempeño computacional de modelos de simulación. Entre ellos destacan técnicas avanzadas de paralelización de código, optimización en el manejo de estructuras de datos, mejoras en la compilación y generación de código ejecutable, aprovechamiento de hardware especializado (por ejemplo, GPU o procesadores multinúcleo) y refactorización lógica de los modelos.

No obstante, la reutilización y el impacto potencial de estos enfoques suelen estar limitados por su especificidad. Por un lado, aquellos esfuerzos enfocados exclusivamente en las particularidades de un único modelo tienen un alcance naturalmente restringido. Por otro lado, cuando las mejoras de rendimiento se aplican a algoritmos genéricos de simulación, no siempre es posible garantizar que tales modificaciones sean beneficiosas para todos los tipos de modelos compatibles con dichos algoritmos. Este dilema plantea la necesidad de adoptar enfoques que permitan optimizar el rendimiento de manera generalizada, sin comprometer la flexibilidad del simulador.

1.3 Ventajas del formalismo DEVS y del simulador PowerDEVS

En este contexto, los simuladores que se adhieren a formalismos que separan claramente el concepto de *modelo* del concepto de *simulador* utilizado para ejecutarlo ofrecen ventajas relevantes. Entre estos formalismos destaca el formalismo DEVS (*Discrete EVent System specification*), propuesto por Bernard Zeigler [36], que permite representar de manera exacta cualquier sistema de tiempo y eventos discretos con gran generalidad, además de ofrecer aproximaciones de sistemas continuos con precisión controlada mediante técnicas de cuantización de estados [8]. DEVS se caracteriza por su flexibilidad, permitiendo modelar tan-

to sistemas determinísticos como estocásticos [11], y por su capacidad de actuar como denominador común para una gran variedad de formalismos matemáticos, los cuales pueden representarse de manera equivalente bajo DEVS [32].

Diversas herramientas han sido creadas para construir y simular modelos DEVS, utilizando distintos lenguajes de programación y enfocadas en diferentes tipos de modelos y/o usuarios [31]. En particular, el simulador PowerDEVS [2] es una herramienta basada en el formalismo DEVS que posee más de doce años de evolución continua. PowerDEVS cuenta con una amplia biblioteca de componentes pre-desarrollados que abarcan múltiples disciplinas, desde la ingeniería y la biología hasta las redes de comunicaciones y el análisis de comportamientos sociales, así como una interfaz gráfica que facilita su uso para usuarios no expertos. La existencia de esta extensa biblioteca y la gran cantidad de modelos desarrollados a lo largo del tiempo hacen que cualquier mejora en el desempeño del motor de simulación pueda potencialmente beneficiar a una gran comunidad de usuarios y aplicaciones.

La separación estricta entre la especificación del modelo y el motor de simulación en DEVS, y en particular en PowerDEVS, asegura que las mejoras introducidas en un modelo específico no afectan al rendimiento de otros modelos. Asimismo, las mejoras aplicadas al motor de simulación (común a cualquier modelo) tienen el potencial de beneficiar, en mayor o menor medida, a todos los modelos compatibles con dicho motor. Esta característica constituye una ventaja significativa respecto a enfoques más específicos o ad-hoc, ya que permite una mejor reutilización y generalización de las mejoras introducidas.

1.4 Objetivos y alcance de esta tesis

Dadas estas consideraciones, el **objetivo general** de esta tesis consiste en proponer, desarrollar e implementar mejoras significativas en el rendimiento del simulador PowerDEVS, manteniendo la generalidad y flexibilidad del formalismo DEVS. Para lograr esto, se adoptará una metodología robusta y reproducible que permita evaluar cuantitativamente el impacto de dichas mejoras sobre una gran variedad de modelos conocidos existentes. Adicionalmente, se buscará po-

der explicar detalladamente las causas técnicas detrás de las mejoras observadas, analizando cómo estas modificaciones interactúan con la arquitectura interna del software intervenido.

En particular, esta tesis abordará:

- La identificación y análisis crítico de cuellos de botella en el desempeño actual de PowerDEVS (medida como el tiempo de ejecución de una simulación).
- El rediseño e implementación de mejoras en aspectos clave del simulador, como el manejo eficiente de estructuras de datos internas, optimización de rutinas críticas y técnicas avanzadas de compilación.
- La definición de una metodología sistemática de evaluación del rendimiento, que permita medir objetivamente los efectos de las modificaciones realizadas sobre un conjunto representativo de modelos.
- La extensión de un *benchmark* de comparación de rendimiento de uso común en simuladores DEVS.
- La validación experimental de las mejoras propuestas, demostrando su aplicabilidad general y cuantificando sus beneficios en términos de eficiencia computacional.

Con este enfoque, se busca contribuir no solo a mejorar el desempeño específico del simulador PowerDEVS, sino también a establecer una base metodológica sólida para futuras investigaciones en optimización del rendimiento en simuladores basados en DEVS. Los resultados de este trabajo de tesis fueron publicados en un trabajo de conferencia internacional [24].

1.5 Síntesis de Contribuciones Principales

La siguiente es una lista que resume las contribuciones originales presentadas en esta tesis:

1. Optimización del motor de simulación PowerDEVS para mejorar el rendimiento en términos de tiempo de ejecución.

2. Desarrollo de herramientas de profiling para detectar potenciales fugas de rendimiento y para comparar automáticamente el consumo de recursos computacionales entre versiones del software.

3. Implementación de un sistema de logging eficiente en el núcleo del simulador, con niveles configurables de verbosidad.

4. Aplanamiento automático de modelos para simplificar la estructura jerárquica y reducir la complejidad en la mensajería interna.

5. Mejoras en el manejo de conexiones entre modelos, optimizando estructuras de datos para acelerar la propagación de eventos.

6. Cambios en el sistema de scheduling de eventos, incluyendo la implementación de colas de prioridad alternativas (Pairing Heap y Splay Tree).

7. Optimización del integrador QSS3 mediante el reemplazo de funciones numéricas para el cálculo de raíces cúbicas, mejorando precisión y eficiencia.

8. Mejoras en modelos atómicos de DataLoggers, utilizando estructuras de datos más eficientes para el cálculo de métricas estadísticas.

9. Adaptación de modelos de redes de datos para reducir la sobrecarga en el registro de variables, mejorando el rendimiento en simulaciones a gran escala.

10. Validación experimental con casos de estudio reales, como el sistema TDAQ de ATLAS en el CERN, demostrando aceleraciones relativas superiores al 50 %.

1.6 Estructura del documento

El resto de esta tesis se organiza de la siguiente manera: en el Capítulo 2 se presentan los fundamentos teóricos necesarios sobre simulación basada en eventos discretos, el formalismo DEVS y la arquitectura general del simulador PowerDEVS. En el Capítulo 3 se describe en detalle la metodología propuesta para la identificación de cuellos de botella en el desempeño, la implementación de mejoras y la evaluación experimental. En el Capítulo 4 se presentan los resultados obtenidos, junto con un análisis crítico sobre la efectividad de las mejoras implementadas. Finalmente, en el Capítulo 5 se discuten las conclusiones obtenidas, así como posibles líneas futuras de investigación derivadas del presente

trabajo.

2

Conceptos Preliminares

2.1 Formalismo de Especificación de Sistemas de Eventos Discretos (DEVS)

El formalismo de Modelado y Simulación (M&S) DEVS [36] es un marco formal para la descripción de modelos de eventos discretos equipado con un algoritmo de simulación abstracto que es independiente de la implementación del modelo.

DEVS es el formalismo más general para representar sistemas de eventos discretos, y permite representar de manera exacta cualquier sistema de tiempo y eventos discretos. DEVS permite también aproximar numéricamente sistemas continuos con cualquier grado de precisión deseado mediante los métodos numéricos QSS [8], que se pueden representar naturalmente con DEVS. Este último concepto será abordado en detalle en la Sección 2.2.2 con los métodos QSS.

Desde su formulación original se han probado muchas propiedades de DEVS, incluyendo el homomorfismo con otros formalismos que pueden ser representados por modelos DEVS [32], tales como autómatas celulares, máquinas de estados finitos, redes de Petri, Bond-Graph, etc.

DEVS es capaz de representar cualquier sistema cuyo comportamiento de entrada/salida pueda describirse mediante secuencias de eventos, siempre que

en cualquier intervalo de tiempo finito se tenga un número finito de eventos. De lo contrario, dicho modelo DEVS se denomina *ilegítimo* y no puede simularse.

Un sistema modelado con DEVS está definido por una composición modular y jerárquica de submodelos, que pueden ser de *comportamiento* (atómicos) o *estructurales* (acoplados). Estos submodelos interactúan por medio de mensajes enviados a través de sus puertos de entrada/salida, de un modo orientado a la interconexión de bloques.

Como se observa en la Figura 2.1 un modelo DEVS procesa una trayectoria de eventos de entrada $x(t)$ y, de acuerdo con su propio estado interno $s(t)$, produce una trayectoria de eventos de salida $y(t)$. Un *evento* representa un cambio instantáneo en alguna parte del sistema, y se puede caracterizar por su tiempo de ocurrencia y un valor. Se puede observar que los eventos pueden ocurrir en cualquier instante de tiempo, por lo tanto el eje de tiempos es continuo. El valor del evento puede ser un número, una palabra o, en general, cualquier elemento de un conjunto arbitrario X. Una *trayectoria* se define como una secuencia de eventos. Esta toma el valor $\emptyset$ (o SIN EVENTO) excepto en los instantes de tiempo en los que hay eventos. En esos instantes, la trayectoria toma el valor correspondiente al evento [12]. En la Figura 2.1 se pueden observar a modo de ejemplo las trayectorias de eventos de entrada $x(t)$ y de salida $y(t)$.

2.1.1 Modelo DEVS Atómico

En términos de **comportamiento**, un modelo **atómico** DEVS Clásico [1] **A** se define mediante la siguiente tupla:

$$A = \{S, X, Y, \delta_{int}, \delta_{ext}, ta, \lambda\},$$

donde S es el conjunto de estados internos, X es el conjunto de mensajes de entrada aceptados e Y es el conjunto de mensajes de salida disponibles. Existen cuatro funciones dinámicas que definen el comportamiento de un modelo atómico: función de avance de tiempo $ta(s) : S \rightarrow \mathbb{R}_0^+$ es la vida útil de cada

[1] Esta especificación de un modelo atómico DEVS es válida en la especificación original del formalismo desarrollada por B. Zeigler [36] conocida como DEVS *Clásico* o *DEVS Secuencial*.

estado $s \in S$. Después de $ta(s)$ unidades de tiempo, se produce una transición interna de estado $\delta_{int}(s) : S \to S$ (suponiendo que no lleguen eventos de entrada externos). En caso de que llegue un evento externo se dispara la ejecución de la función de transición externa $\delta_{ext}(s, e, x) : S \times \mathbb{R}_0^+ \times X \to S$, siendo e el tiempo transcurrido en el estado s con $0 \leq e < ta(s)$. Cada vez que se calcula un nuevo estado s' (ya sea invocando δ_{int} o δ_{ext}), se calcula un nuevo tiempo de vida $ta(s')$ y el tiempo transcurrido e se restablece a 0. Finalmente, la función de salida $\lambda(s) : S \to Y$ se ejecuta luego de transcurrido un tiempo $ta(s)$ desde el último evento, e inmediatamente antes de una transición interna, para emitir eventos de salida en Y. Notar que esta función utiliza el valor del estado antes de la transición interna. Una transición externa se dispara cada vez que se recibe una entrada en X. Para describir el estado total de un modelo atómico no alcanza con $s \in S$ sino que es necesario considerar también el tiempo e transcurrido desde que se transicionó al estado s, entonces se define el *estado total* $Q = \{(s, e) \big| s \in S, 0 \leq e \leq ta(s)\}$. En la Figura 2.1 se observa una representación gráfica de un atómico DEVS que resume todas las funciones mencionadas y su relación con las entradas $x \in X$, las salidas $y \in Y$ y los estados $s \in S$.

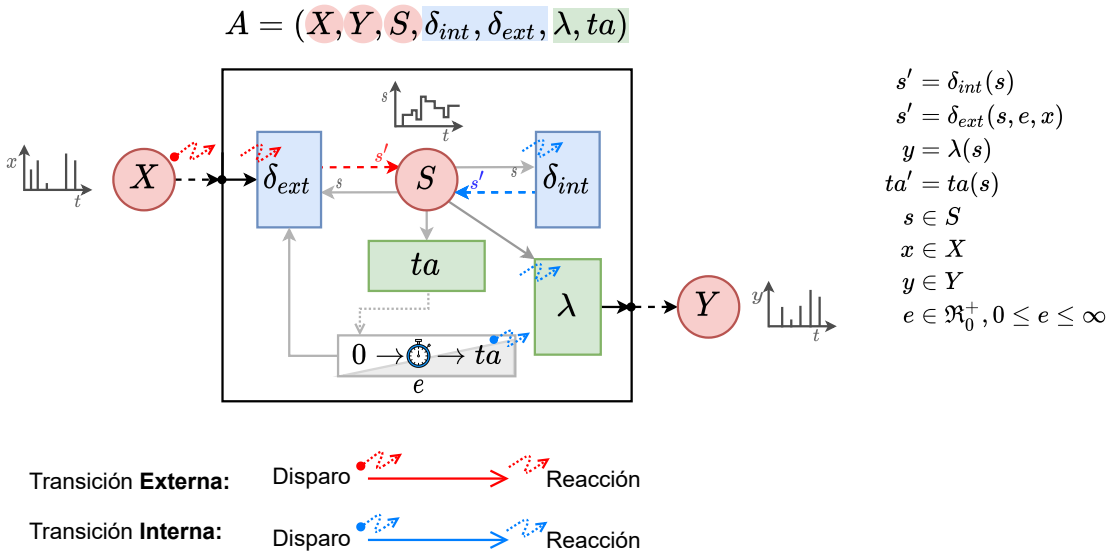


Figura 2.1: Modelo Atómico DEVS A. δ_{int}: función de transición interna, δ_{ext}: función de transición externa, ta: función de avance de tiempo, λ: función de salida. En rojo, los conjuntos X, Y y S, de mensajes de entrada, mensajes de salida y estados respectivamente. En celeste, las funciones de transición interna y externa. Y en verde, las funciones de salida y avance de tiempo.

A partir de lo anterior, podemos notar que DEVS es por definición un formalismo asíncrono, dado que cada modelo atómico controla su propio reloj (avance de tiempo $ta(s)$) de forma independiente de los demás. En la Sección 2.1.4 veremos que cada modelo atómico tiene asociada una rutina del simulador para el manejo de este tiempo.

En la Figura 2.2 se puede observar un ejemplo de la trayectoria que sigue el estado de un modelo atómico DEVS. Se puede apreciar que el atómico comienza en el estado s_1 en el instante t_1 y permanece en ese estado por un tiempo $ta(s_1)$. Cuando se extingue este tiempo (en $t_1 + ta(s_1)$) se produce un evento de salida con valor $y_1 = \lambda(s_1)$. En el mismo instante de tiempo se produce una **transición interna** evolucionando al estado $s_2 = \delta_{int}(s_1)$, en el que se permanecerá por un tiempo $ta(s_2)$ y se restablece el tiempo e. Sin embargo, este último tiempo es interrumpido por la aparición de un evento externo x_1 que dispara una transición instantánea hacia un nuevo estado definido por la función de **transición externa** $s_3 = \delta_{ext}(s_2, e, x_1)$, que depende del valor del evento x_1, del estado del modelo en el momento en que se produce el evento (s_2) y del tiempo que ya permaneció en ese estado ($e \leq ta(s_2)$). Notar que no se produce ningún evento de salida durante una transición externa. El modelo permanecerá en el estado s_3 durante un tiempo $ta(s_3)$, tras el cual se producirá un evento de salida de valor $y_2 = \lambda(s_3)$ y simultáneamente se disparará una transición interna evolucionando al estado $s_4 = \delta_{int}(s_3)$.

Con la incorporación de puertos de entrada y salida a este modelo atómico se contribuye a simplificar las tareas de modelado. De este modo, los eventos de entrada y salida tienen ahora asociado un puerto: $X = \{(p.x)|p \in InPorts, x \in X_p\}$ e $Y = \{(p.y)|p \in OutPorts, y \in Y_p\}$, respectivamente.

2.1.2 Modelo DEVS Acoplado

En términos de **estructura**, un modelo **acoplado** DEVS **C**, como el de la Figura 2.3, define la estructura del modelo, es decir, cómo se interconectan los componentes atómicos a través de sus puertos de entrada/salida. Formalmente,

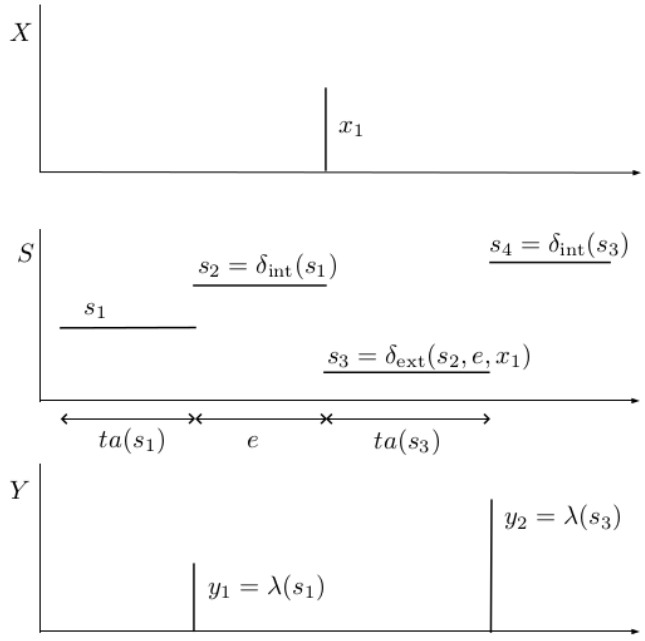


Figura 2.2: Trayectorias en un modelo DEVS (tomado con permiso de [12]).

un modelo acoplado DEVS Clásico [2] se puede describir mediante la siguiente tupla:

$$C = \{X_C, Y_C, D, M_d | d \in D, EIC, EOC, IC, Select\} \tag{2.1}$$

donde:

- $X_C = \{(p,x) | p \in InPorts, x \in X_p\}$ e $Y_C = \{(p,y) | p \in OutPorts, y \in Y_p\}$ son los conjuntos de mensajes de entrada/salida y de puertos de entrada/-salida del modelo acoplado respectivamente,
- D es el conjunto de nombres de los componentes del acoplado C, tal que para cada $d \in D$, M_d es un modelo atómico DEVS ($M_d = \{S, X_d, Y_d, \delta_{int}, \delta_{ext}, ta, \lambda\}$) o un modelo acoplado DEVS ($M_d = \{X_d, Y_d, D', M_{d'} | d' \in D', EIC, EOC, IC, Select\}$), en ambos casos con $X_d = \{(p,x) |\, p \in InPorts_d,\, x \in X_p\}$ e $Y_d = \{(p,y) |\, p \in OutPorts_d,\, y \in Y_p\}$,
- Los *acoplamientos de entradas externas* (EIC) definen las conexiones entre los puertos de entrada del modelo acoplado C con los puertos de entrada de

[2]Este modelo Acoplado corresponde a DEVS Clásico, debido a la presencia de la función *Select* que se elimina en otras variantes del formalismo, e.g. Parallel DEVS [36].

los componentes: $EIC \subseteq \{((C, ip_C), (d, ip_d)) | \, ip_C \in InPorts, \, d \in D, \, ip_d \in InPorts_d\}$

- Los *acoplamientos de salidas externas* (EOC) definen las conexiones entre los puertos de salida de los componentes con los puertos de salida de C: $EOC \subseteq \{((d, op_d), (C, op_C)) | \, op_C \in OutPorts, \, d \in D, \, op_d \in OutPorts_d\}$

- Los *acoplamientos internos* (IC) definen las conexiones entre las salidas de componentes con entradas de componentes: $IC \subseteq \{((a, op_a), (b, ip_b)) | \, a, b \in D, \, op_a \in OutPorts_a, \, ip_b \in InPorts_b\}$. Sin embargo, no están permitidos los lazos directos, es decir, ningún puerto de salida de un componente puede estar conectado a un puerto de entrada del mismo componente: $((d, op_d), (e, ip_d)) \in IC$ implica que $d \neq e$.

- $Select : 2^D \to D$, es la *función de desempate* entre dos eventos simultáneos y debe cumplir $Select(E) \in E$, con $E \subset 2^D$ y donde 2^D es el subconjunto de componentes que produce los eventos simultáneos.

El rol de la función *Select* (en DEVS Clásico) es clave cuando dos o más componentes tienen agendada una transición interna en el mismo instante de tiempo. Dado que las transiciones internas pueden disparar eventos de salida que a su vez afecten a sus *influenciados*, la elección del orden de ejecución no es trivial y puede afectar de forma significativa al comportamiento del sistema.

En ejemplo de la Figura 2.3 tenemos un modelo acoplado C que contiene dos atómicos M_a y M_b. En este caso, tenemos el conjunto de nombres de componentes de C dado por $D = \{a, b\}$, el conjunto de componentes $M_d = \{M_a, M_b\}$, y los conjuntos de acoplamientos de entradas externas $EIC = \{((C, ip_C), (a, ip_a))\}$, acoplamientos de salidas externas $EOC = \{((a, op1_a), (C, op1_C)), ((b, op1_b), (C, op2_C))\}$ y acoplamientos internos $IC = \{((b, op1_b), (a, ip_a)), ((a, op2_a), (b, ip_b))\}$.

2.1.3 Clausura Bajo Acoplamiento DEVS

El formalismo DEVS es *cerrado bajo acoplamiento*, es decir: cualquier composición jerárquica de modelos atómicos y acoplados DEVS (como en la Figura 2.3) define un modelo DEVS atómico equivalente. Más aún, la composición de varios

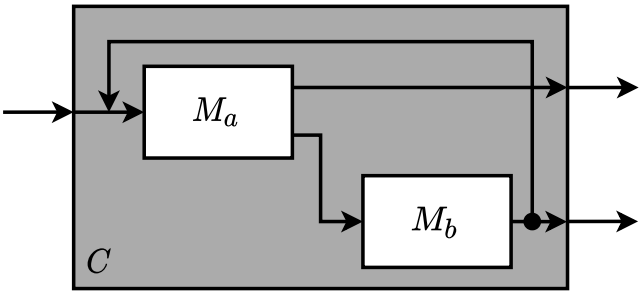


Figura 2.3: Modelo Acoplado DEVS, con $D = \{a, b\}$ y $M_d = \{M_a, M_b\}$.

modelos atómicos en un modelo acoplado más grande conserva su independencia temporal (a través de su función de avance de tiempo).

Dado que DEVS es un formalismo modular y jerárquico, es naturalmente apto para representar sistemas diseñados con estructuras jerárquicas. Esto se debe a su propia estructura basada en modelos atómicos y acoplados, que sólo pueden comunicarse a través de eventos vía sus puertos de entrada/salida. El estado interno de cada modelo atómico permanece oculto a otros modelos, y la única forma de conocer y/o modificar su estado es mediante el intercambio de eventos. Este hecho es de suma importancia, debido a la rigurosidad de las pruebas formales de sus propiedades [33].

2.1.4 Simulador Abstracto DEVS

Hasta aquí estudiamos cómo construir modelos bajo el formalismo DEVS. En esta sección presentaremos cómo el mismo formalismo define la manera de simular estos modelos.

La simulación en DEVS se realiza por medio de un *Simulador Abstracto* cuya especificación es independiente del lenguaje de programación con que se implemente. En esencia, el simulador abstracto de DEVS se puede ver como un *planificador* de eventos que definen la ejecución en el tiempo de los modelos atómicos y acoplados que componen a un modelo DEVS.

La idea básica para la simulación de un modelo DEVS acoplado se puede describir mediante los siguientes pasos [12]:

1. Buscar entre sus subordinados (d en la Ecuación 2.1) el modelo atómico d^* que, de acuerdo a su tiempo de avance y al tiempo transcurrido, sea

el próximo en realizar una transición interna. Llamaremos *modelo atómico inminente* a d^* y t_n al tiempo de esta transición.

2. Avanzar el tiempo de la simulación t hasta $t = t_n$ y ejecutar la función de transición interna de d^*.

3. Propagar el evento de salida producido por d^* a todos los modelos atómicos conectados a este (conjunto de *modelos influenciados* de d^*), ejecutando sus correspondientes funciones de transición externa. Luego, volver al paso 1.

Una forma sencilla de implementar estos pasos es mediante un programa con una estructura jerárquica equivalente a la estructura jerárquica del modelo a simular. Cada modelo atómico se asocia con una rutina llamada *Simulador-DEVS*, mientras que una rutina *Coordinador-DEVS* se asocia a cada modelo acoplado. En la cima de esta estructura hay una rutina *Coordinador Raíz DEVS* que es la encargada de avanzar el tiempo global de la simulación. La Figura 2.4b muestra el simulador abstracto asociado con el modelo DEVS de la Figura 2.4a.

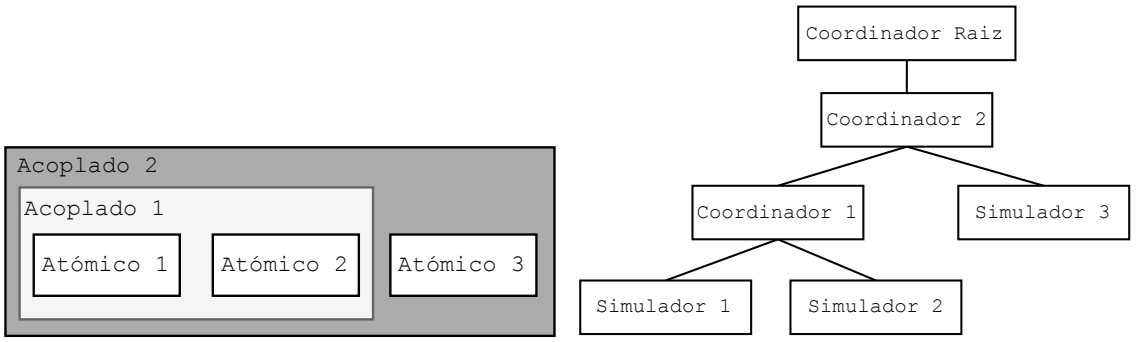


Figura 2.4: (a) Modelo Acoplado DEVS y (b) Simulador Abstracto DEVS asociado.

Cada *Simulador* y *Coordinador* del modelo tiene una variable local t_n que indica el instante de tiempo en el que ocurrirá la próxima transición interna. En un *Simulador*, t_n se calcula utilizando la función de avance de tiempo *ta* del atómico asociado. En un *Coordinador*, t_n es el mínimo t_{n_d} de entre sus modelos subordinados $d \in D$. Entonces el t_n^* del *Coordinador* más alto en la jerarquía (Coordinador 2 en la Figura 2.4b) es el instante de tiempo en el que ocurrirá el próximo evento

del modelo global. El *Coordinador Raíz* es el responsable de avanzar el tiempo global de la simulación t a ese valor: $t = t_n^*$. De esta manera, en DEVS *Clásico* la simulación de un modelo por eventos discretos consiste en el agendamiento y ejecución de eventos que se ejecutan de manera secuencial provocando a su vez nuevos eventos con cada ejecución que serán agregados a la lista.

Una desventaja de esta implementación del simulador siguiendo una estructura jerárquica es la gran cantidad de tráfico de *mensajes del simulador abstracto* que podría generarse entre distintas capas. Por ejemplo, en la Figura 2.4b podemos ver que el envío de un evento desde el Atómico 1 al Atómico 3, implica el intercambio de mensajes del Simulador 1 con el Coordinador 1, del Coordinador 1 con el Coordinador 2 y del Coordinador 2 con el Simulador 3. Cuantos más niveles existan en el simulador abstracto (asociado con los niveles en el modelo de simulación) mayor cantidad de mensajes se intercambiarán sumando carga computacional. Esto podría evitarse ubicando a todos los *Simuladores* en un mismo nivel bajo un único *Coordinador*. Esto corresponde a una *estructura de simulación plana*. Existen algoritmos para producir automáticamente modelos equivalentes aplicando técnicas de *aplanamiento de jerarquías*.

2.2 Resolución Numérica de Ecuaciones Diferenciales Ordinarias

El comportamiento de un sistema dinámico de variables continuas se describe mediante un conjunto de ecuaciones diferenciales [12]. Cuando se trabaja con sistemas físicos de parámetros concentrados y sin retardos se utiliza un conjunto de Ecuaciones Diferenciales Ordinarias (ODEs), mientras que para sistemas dinámicos de parámetros distribuidos o con retardos se obtienen Ecuaciones en Derivadas Parciales (PDEs) o Ecuaciones Diferenciales con Retardo (DDEs) respectivamente.

Siempre que sea posible, la mejor opción es resolverlas y trabajar con la solución analítica exacta. Sin embargo, en general no es posible encontrar soluciones analíticas para estos sistemas o bien es poco práctico, y por lo tanto necesitamos recurrir a métodos numéricos de integración y simulación para aproximar estas

soluciones. A continuación presentaremos algunos elementos para la resolución aproximada de ODEs. Para lidiar con PDEs contamos con distintos métodos para aproximarlos con ODEs (e.g. Método de Líneas (MOL) o los Métodos Lax-Wendroff, por mencionar solo algunos) y luego resolverlos con alguno de los métodos que se utilizan para resolver ODEs.

En lo que sigue consideraremos un sistema dinámico modelado con una ODE como en la Ecuación 2.2, que corresponde con una ecuación de estados con $\mathbf{x}(t) = [x_1(t), \ldots, x_n(t)]^T \in \mathbb{R}^n$ vector de estados de dimensión n y $\mathbf{u}(t) = [u_1(t), \ldots, u_m(t)]^T \in \mathbb{R}^m$ vector de entradas de dimensión m:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \\ \mathbf{x}(t = t_0) &= \mathbf{x}_0, \end{aligned} \tag{2.2}$$

donde: $\dot{\mathbf{x}}(t)$ es la derivada del vector de estados con respecto al tiempo, $\mathbf{x}_0$ es el vector de condiciones iniciales que determina dónde arrancan las trayectorias solución y t es el tiempo. Cada elemento $x_i(t)$ del vector de estados representa la trayectoria del estado i como una función del tiempo. Siempre y cuando $f_i(\mathbf{x}(t), \mathbf{u}(t))$ ni ninguna de sus derivadas contenga discontinuidades entonces $x_i(t)$ será una función continua del tiempo.

A continuación abordaremos algunos métodos numéricos para la aproximación de la solución de sistemas de ecuaciones diferenciales.

2.2.1 Métodos Numéricos Clásicos a Tiempo Discreto

Los métodos de integración convencionales se basan en la discretización del tiempo. Dado que $x_i(t)$ es una función continua del tiempo podrá ser aproximada con cualquier grado de exactitud deseado por medio de la expansión en Serie de Taylor alrededor de cualquier punto.

En la Ecuación 2.2 vimos que el estado $\dot{x}_i = f_i(\mathbf{x}(t), \mathbf{u}(t))$ es función de los mismos estados, las entradas y el tiempo. Dado que nos interesa hallar métodos para obtener una aproximación de las trayectorias temporales entonces a continuación desarrollaremos la expansión en Serie de Taylor del estado $x_i(t)$ en el

tiempo t^* alrededor del que queremos aproximar la solución. Sea $t^* + h$ el punto en donde queremos evaluar esa aproximación, el valor de la trayectoria del estado x_i en este punto se calcula como:

$$x_i(t^* + h) = x_i(t^*) + \frac{dx_i(t^*)}{dt} \cdot h + \frac{d^2x_i(t^*)}{dt^2} \cdot \frac{h^2}{2!} + \dots \tag{2.3}$$

donde h es el paso de integración. Incorporando la Ecuación 2.2 en esta última:

$$x_i(t^* + h) = x_i(t^*) + f_i(\mathbf{x}(t^*), \mathbf{u}(t^*)) \cdot h + \frac{df_i(\mathbf{x}(t^*), \mathbf{u}(t^*))}{dt} \cdot \frac{h^2}{2!} + \dots \tag{2.4}$$

Esto último es el punto de partida de una gran cantidad de métodos numéricos explícitos de aproximación de soluciones de ODEs que se caracterizan por la cantidad de términos que consideran en la expansión de la serie así como por el método de aproximación de las derivadas de f_i. Existe una vasta literatura sobre métodos numéricos con distintas características que los hacen adecuados para abordar distintos tipos de problemas. Para una buena introducción al tema se puede consultar [12]. El error de la aproximación depende tanto del truncamiento de la serie como del error de la aproximación de las derivadas. Muchos métodos numéricos de paso variable estiman este error y usan esa información para ajustar el paso de integración h. Asimismo, se define un juego de parámetros para el control del error de la aproximación que influyen en el cálculo del paso de integración, destacándose las tolerancias relativa y absoluta.

Los métodos numéricos clásicos, tanto los de paso fijo como los de paso variable, discretizan el tiempo de la ODE original (Ecuación 2.2). Por lo tanto transforman un sistema de tiempo continuo en uno de tiempo discreto, en donde sólo conocemos el valor de los estados en los instantes de muestreo, y entre dos instantes de muestreo consecutivos no conocemos qué valores toma la aproximación de la trayectoria. Cuando trabajamos con sistemas dinámicos híbridos la evolución de las variables de estado de tiempo continuo se ve interrumpida por la aparición de eventos discretos que generan discontinuidades en las trayectorias. La base de tiempo discreta elegida para aproximar una ODE no necesariamente coincide con los instantes de tiempo en que se producen estos

eventos. En este caso, los métodos numéricos de tiempo discreto deben incurrir en algoritmos iterativos costosos en términos computacionales para detectar la discontinuidad, resincronizar el tiempo y reiniciar la simulación desde allí. Esta estrategia no escala bien cuando se trabaja con sistemas donde se producen una gran cantidad de discontinuidades (estrictamente hablando: eventos discretos) [22, 8].

Esto último permite concluir que los métodos de integración clásicos tienen dificultades para tratar con la simulación de sistemas híbridos. En la siguiente sección abordaremos una familia de métodos de integración numérica que plantean el problema de integración de una forma alternativa, que presentan ventajas para lidiar con sistemas híbridos y en general conducen a resoluciones más eficientes en términos computacionales.

2.2.2 Métodos Numéricos a Eventos Discretos: Integración por Cuantificación de Estados.

Una idea dual a la tradicional de la discretización del tiempo es la cuantización de estados. Esta idea fue sugerida por primera vez por B. Zeigler [35] y luego elaborada por E. Kofman [22] con la adición de una función de histéresis, dando lugar a la familia de algoritmos QSS [22, 8]. De este modo, en lugar de discretizar el tiempo los métodos QSS cuantifican las variables de estado y resuelven las ODEs utilizando aproximaciones de señales continuas por eventos discretos. De esta manera, se aproxima al sistema dinámico de tiempo continuo por uno de eventos discretos.

En este caso, el sistema de la Ecuación 2.2 se aproxima por el *sistema de estados cuantizados*:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{f}(\mathbf{q}(t), \mathbf{u}(t)) \\ \mathbf{q}_0 &= \mathbf{x}_0 \end{aligned} \tag{2.5}$$

donde $\mathbf{q}(t)$ es el *vector de estados cuantizados* que se obtiene a partir de cuantizar cada una de las variables de estado $x_i(t)$ del vector $\mathbf{x}(t)$, y donde cada $q_i(t)$ recibe

el nombre de *variable cuantizada*. Es importante destacar que el vector de estados $\mathbf{x}(t)$ de la Ecuación 2.5 es una aproximación del de la Ecuación 2.2, que se acerca cuanto más pequeño es el *quantum* de la función de cuantización. La relación entre $\mathbf{q}(t)$ y $\mathbf{x}(t)$ en la Ecuación 2.5 está dada por una función de cuantización y una funcion de histéresis definidas por el método QSS que se utilice.

En el método QSS de primer orden (QSS1) se utiliza una función de cuantización de orden cero y en consecuencia cada componente $q_i(t)$ sigue trayectorias constantes a tramos, válidas siempre que la diferencia entre ésta y la variable de estado (aproximada) correspondiente $x_i(t)$ sea menor que el valor del *quantum* ΔQ_i. Luego, si las componentes $u_i(t)$ del vector de entradas $\mathbf{u}(t)$ siguen trayectorias constantes por tramos, las componentes de $\dot{\mathbf{x}}(t)$ en la Ecuación 2.5 seguirán también trayectorias constantes a tramos y en consecuencia los estados $x_i(t)$ siguen trayectorias lineales a tramos. Llamando t_i a los instantes de inicio de cada sección constante de $q_i(t)$, la relación entre la trayectoria cuantizada $q_i(t)$ y la correspondiente trayectoria del estado $x_i(t)$, está dada por:

$$q_i(t) = \begin{cases} q_i(t_k) & \text{si } |x_i(t) - q_i(t_k)| < \Delta Q_i \\ x_i(t) & \text{en otro caso,} \end{cases} \tag{2.6}$$

para $t_k \leq t < t_{k+1}$, donde t_{k+1} es el primer instante de tiempo en el que se cumple $|x_i(t) - q_i(t_k)| = \Delta Q_i$. La Ecuación 2.6 corresponde a la cuantización de orden 0 que se observa en la Figura 2.5. La función de histéresis (con valor ϵ), dada por la función módulo en la Ecuación 2.6, se incorpora para evitar el problema de *chattering*. El efecto de la histéresis se observa en la Figura 2.6 (para el caso de QSS1 y eligiendo $\epsilon = \Delta Q$) por ejemplo cuando $x(t)$ cruza el valor de $q(t)$ en $t = t^*$ instante en el cual *no se produce* un cambio en el estado cuantizado $q(t)$ (x debería alcanzar el valor $q - \Delta Q$). En la Figura 2.7 se observa un conjunto de n integradores QSS1 conformando el sistema de la Ecuación 2.5.

Los métodos QSS de orden superior utilizan esta misma idea de $q_i(t)$ y $x_i(t)$ siguiendo trayectorias polinomiales de a tramos de orden n y $n+1$ respectivamente, modificando la función de cuantización. Por ejemplo, en el método QSS de segundo orden (QSS2) se utiliza un cuantizador de primer orden que genera trayectorias de salida que son lineales a tramos, cuya pendiente y ordenada al

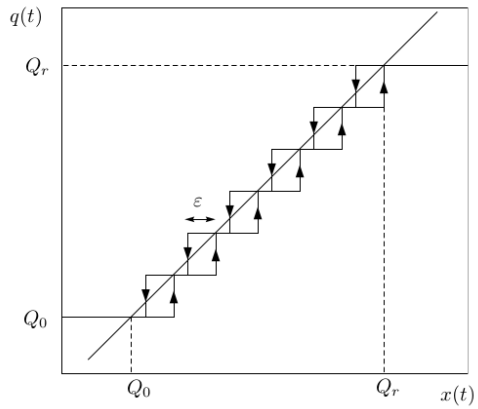


Figura 2.5: Función de Cuantización para QSS1 (tomado de [12] con autorización)

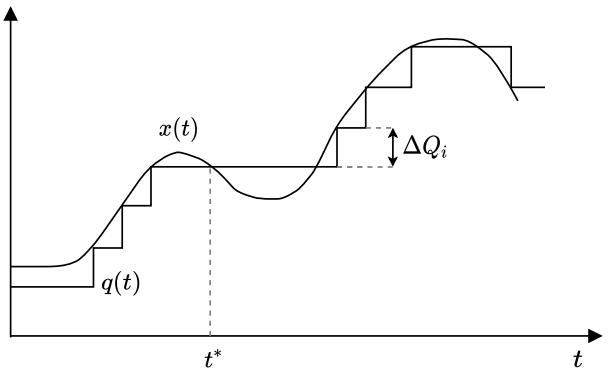


Figura 2.6: Efecto de la histéresis en la cuantización para QSS1.

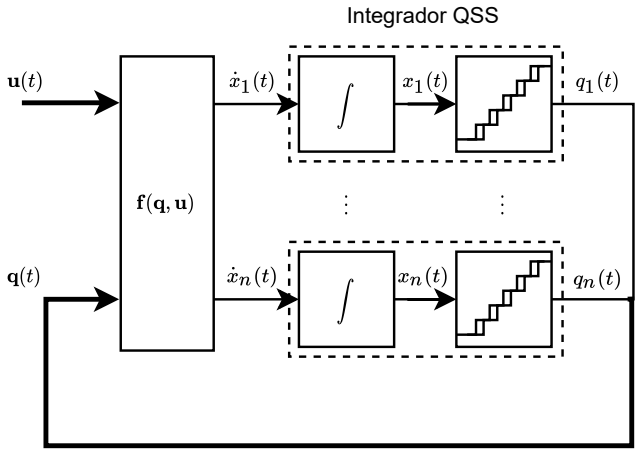
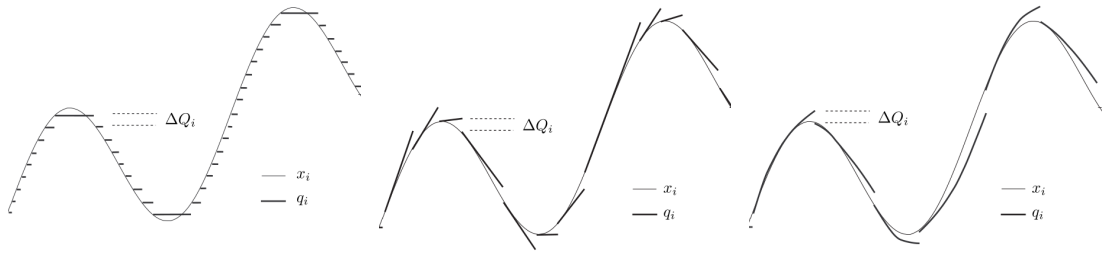


Figura 2.7: Sistema de Estados Cuantizados $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{u}(t))$ con integradores QSS de primer orden (QSS1).

origen cambian cada vez que la diferencia entre la entrada y la salida del cuantizador es mayor al *quantum*. La Ecuación 2.5 sigue siendo válida, pero ahora la función de cuantización a la salida de los integradores es de primer orden. En este caso las trayectorias de $\dot{\mathbf{x}}(t)$ también son lineales a tramos y por lo tanto las trayectorias del estado $\mathbf{x}(t)$ son parábolas (Figura 2.8b).

En la Figura 2.9 se puede observar el funcionamiento del método QSS de segundo orden o QSS2. En el panel (a) se observan las trayectorias de la variable de estado $x(t)$ (curva azul) (uno de los elementos del vector $\mathbf{x}(t)$ en la Ecuación 2.5) y su estado cuantizado correspondiente $q(t)$ (curva verde), compuestas por tramos cuadráticos y lineales, $x(t) = a_0 + a_1 \cdot (t - t_k) + a_2 \cdot (t - t_k)^2$



(a) (b) (c)

Figura 2.8: Trayectorias obtenidas para: (a) QSS, (b) QSS2 y (c) QSS3

y $q(t) = a_0 + a_1 \cdot (t - t_k)$, respectivamente. Los puntos indicados sobre la curva marcan los límites de las secciones en donde cambian los coeficientes de estos polinomios. Por ejemplo el punto ① se debe a una actualización en alguna de las entradas de la ecuación de estado que afecta el cálculo de la derivada $\dot{x}$. Por otro lado, el punto ② ocurre cuando la diferencia entre x y q es mayor al *quantum* en ese momento. En el panel (b) se observa la trayectoria del error $e(t) = q(t) - x(t)$ introducido por el método. Cada vez que la diferencia alcanza un valor mayor al *quantum* ΔQ se produce un evento. En ese momento $q(t)$ se actualiza cuantificando a $x(t)$ dando lugar a un nuevo polinomio para el tramo, y el error se restablece a 0. Este evento se propaga al sistema, forzando la evaluación de las ecuaciones de los estados que dependen de $x(t)$.

El valor del *quantum* ΔQ sigue su trayectoria que se puede ver en el panel (c). Esta trayectoria depende de los parámetros: ΔQ_{rel} y $\Delta Q_{\text{mín}}$ que corresponden con la precisión relativa y absoluta, tales que $\Delta Q = \text{máx}(\Delta Q_{min}, \Delta Q_{rel} \cdot |q|)$. El valor mínimo del *quantum* está dado por $\Delta Q_{\text{mín}}$ que juega el mismo papel que la tolerancia absoluta en los métodos numéricos clásicos. Una ventaja de esta familia de métodos numéricos es que estos parámetros se pueden ajustar de manera independiente para cada integrador QSS. Finalmente, los paneles (d), (e) y (f) muestran las trayectorias de los coeficientes de los polinomios de $x(t)$ y $q(t)$.

2.2.2.1 Propiedades de los Métodos QSS

La familia de métodos numéricos QSS presenta varias ventajas con respecto a su contraparte basada en la discretización del tiempo. A continuación listamos

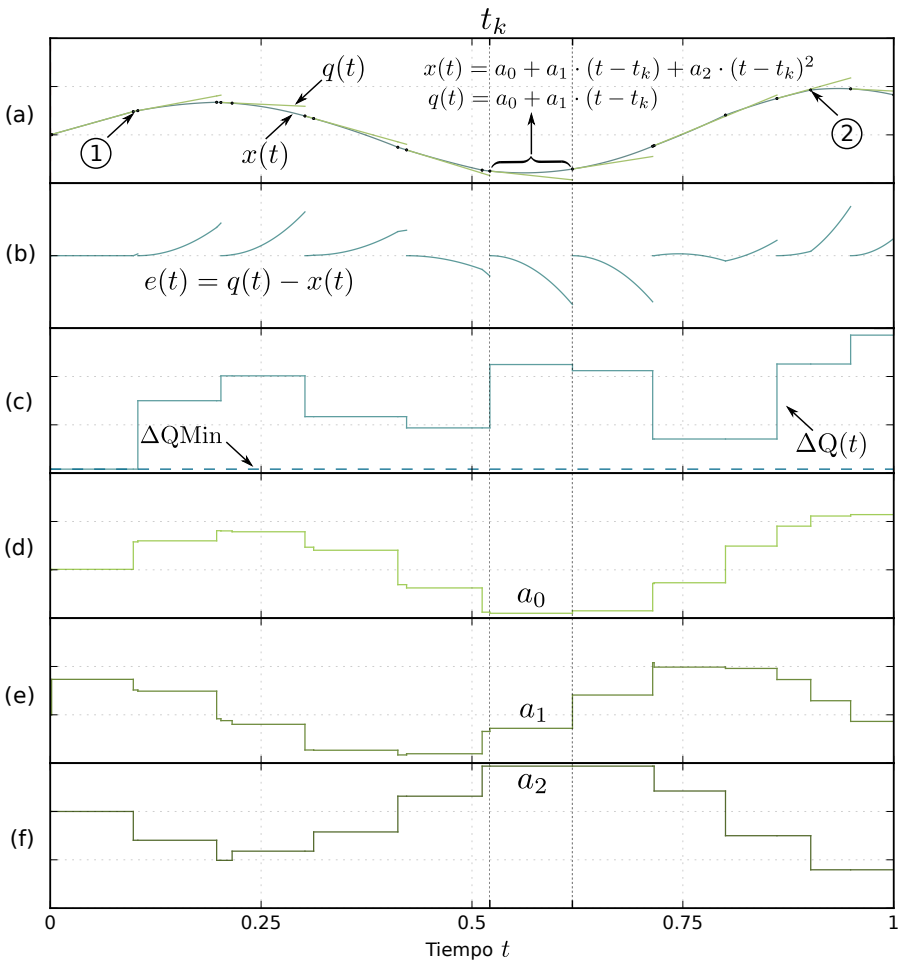


Figura 2.9: Ejemplo ilustrativo de una simulación del sistema $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{u}(t))$ con QSS2 y sus conceptos esenciales relacionados.

las propiedades más destacables:

- **Convergencia** Se demuestra para los métodos QSS [22], QSS2 [19] y QSS3 [20], que cuando el *quantum* ΔQ se elige suficientemente pequeño, las soluciones de la Ecuación 2.5 aproximan a las soluciones de la Ecuación 2.2.

- **Cotas globales de error** Una cota global de error para sistemas lineales de tiempo invariante (LTI) fue estudiada en [19, 21]. El error en una simulación QSS nunca va a diferir de la solución analítica por más que un valor finito proporcional al *quantum* ΔQ. Se puede alcanzar un error de simulación arbitrariamente pequeño cuando se utiliza un ΔQ suficientemente pequeño. Más aún, la cota de error no depende de la condición inicial y permanece constante durante la simulación (a diferencia de los métodos de tiempo discreto).

- **Tiempo de avance asincrónico** Como vimos más arriba cada variable en un sistema cuantizado actualizará su valor en diferentes instantes de tiempo, cada vez que $|x_i(t) - q_i(t_k)| > \Delta Q_i$. Este cambio se propaga luego solo a aquellos estados dependientes, afectando a un subconjunto de los estados del sistema. Esto contrasta con lo que sucede en los métodos clásicos en donde todas las variables se calculan en el mismo paso de integración.

- **Salida densa** La salida de un integrador QSS está representada por una trayectoria polinomial por tramos de acuerdo con el orden del método QSS utilizado (Figura 2.8). En cualquier instante t_0 podemos aproximar a $\mathbf{x}(t_0)$ evaluando el polinomio correspondiente y la diferencia permanece dentro de la cota teórica de error. Esta característica resulta muy útil para sistemas híbridos y asincrónicos. Esto contrasta con la salida de un integrador basado en la discretización del tiempo que solo es válida en ciertos instantes de tiempo.

- **Simulación eficiente de discontinuidades** Dada la naturaleza asincrónica y de eventos discretos de los métodos QSS, cada discontinuidad es manejada de manera natural y eficiente. Por el contrario, los métodos clásicos de tiempo discreto requieren de procedimientos especiales para ejecutar un paso en el momento exacto en el que ocurre una discontinuidad. Esto provee a los métodos QSS con ventajas intrínsecas para simular sistemas discontinuos [13].

Para finalizar, podemos notar que los métodos de integración QSS dan como resultado *una aproximación a eventos discretos de una ODE*. Esto contrasta con los modelos de tiempo discreto que proveen los métodos numéricos de integración clásicos. Luego, esta aproximación QSS puede ser simulada de forma exacta por un modelo DEVS [22].

2.3 PowerDEVS

PowerDEVS [2] es una herramienta para la construcción y simulación de modelos DEVS particularmente adecuada para la simulación eficiente de sistemas

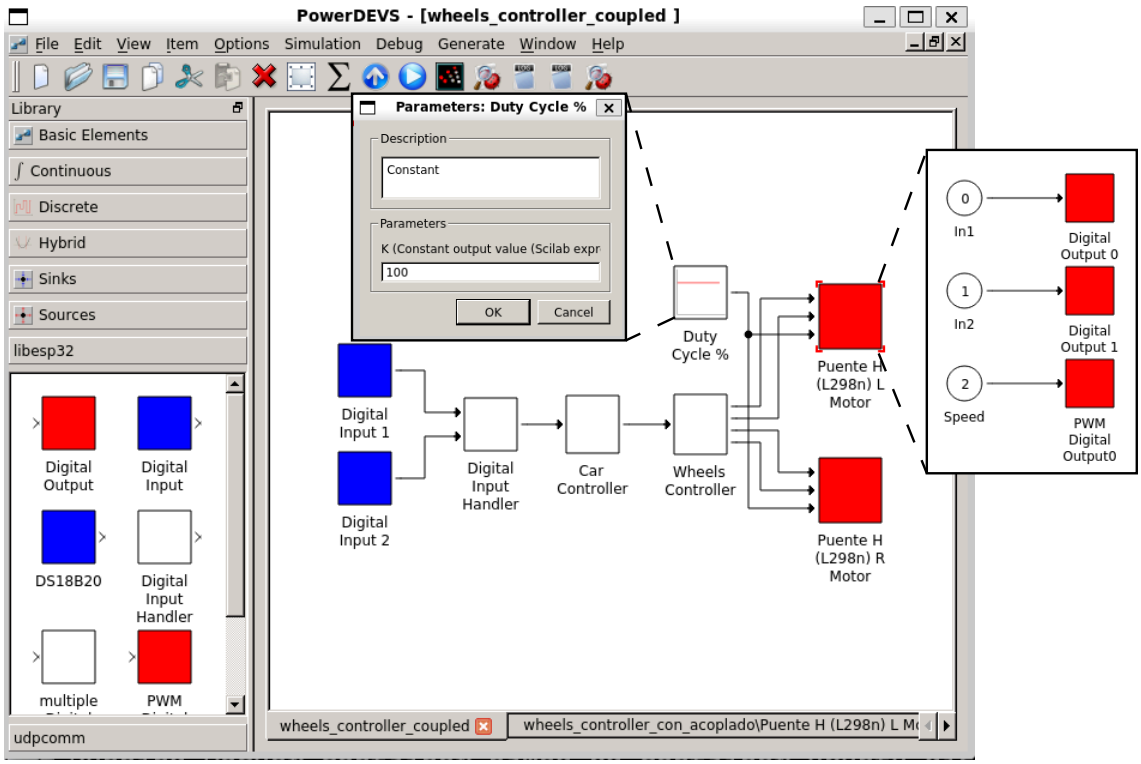


Figura 2.10: Editor de Modelos (GUI) de PowerDEVS. Los bloques gráficos representan modelos atómicos y acoplados. La barra de la izquierda muestra las bibliotecas desde donde se pueden arrastrar, soltar e interconectar modelos por medio de líneas dirigidas que representan el sentido de los eventos.

dinámicos híbridos. Esto se debe en parte a que es el simulador DEVS insignia para la familia de métodos numéricos QSS [8, 9] presentados en la Sección 2.2.2. Una ventaja distintiva de PowerDEVS es su rendimiento en comparación con otros simuladores DEVS (lo que lo hace recomendable para aplicaciones donde el rendimiento de la simulación es un aspecto clave) y ha sido recomendado para modeladores no familiarizados con DEVS (gracias a su interfaz gráfica) y para usuarios que desean utilizar DEVS para la simulación de sistemas híbridos [31].

PowerDEVS [2] consta de cuatro programas independientes principales. El *Editor de Modelos* (`pdme`) es la interfaz gráfica de usuario (GUI) que proporciona un espacio de trabajo para construir y configurar el modelo de simulación arrastrando, soltando e interconectando bloques gráficos asociados a modelos atómicos y acoplados. El resultado es un archivo `pdm` que contiene tanto información gráfica como de parámetros de configuración de los modelos y enlaces

al código a ejecutar. El *Editor de Modelos Atómicos* (`pdae`) está diseñado para describir el comportamiento de los modelos atómicos, es decir, editar el código C++ a ejecutar durante la inicialización y la ejecución de las funciones dinámicas δ_{int}, δ_{ext}, λ y ta. El *Pre-procesador* (`pdppt`) genera código C++ a partir de la representación del modelo gráfico (archivo `pdm`), que luego se compila para producir un ejecutable independiente que incluye la especificación del modelo y el motor de simulación. Finalmente, la *Interfaz de Simulación* (`pdif`) es una interfaz gráfica de control de simulación que permite la ejecución interactiva de la simulación. El archivo ejecutable de la simulación también puede ser ejecutado sin la *Interfaz de Simulación* desde la línea de comandos.

En la *Vista de Diseño*, un modelo se construye con el *Editor de Modelos* interconectando representaciones gráficas de modelos DEVS atómicos y acoplados, desde los que se puede acceder a sus parámetros y a sus submodelos (Figura 2.10). Las líneas dirigidas unen puertos de salida con puertos de entrada y representan el sentido de los eventos. En la barra de la izquierda se observan las bibliotecas desde donde se pueden arrastrar y soltar los modelos disponibles. La *Vista de Simulación* permite establecer los parámetros de simulación (por ejemplo, tiempo final de simulación, número de ejecuciones, etc.) y verificar la evolución del tiempo de simulación a través de la *Interfaz de Simulación* o mediante la línea de comandos con el ejecutable producido por el *Pre-procesador*.

2.3.1 Extensión para Construir Sistemas Complejos y a Gran Escala con Python: py2PowerDEVS

Una alternativa a la GUI para la construcción de modelos de PowerDEVS es utilizar un lenguaje de scripting. `py2PowerDEVS` [25] es una extensión de PowerDEVS que amplía sus capacidades para construir modelos topológicamente grandes utilizando scripts en Python. Esencialmente, esta interfaz de Python a PowerDEVS permite acceder directamente desde código Python a las clases de PowerDEVS (para los modelos atómicos y el motor) definidas en C++, mientras que la ejecución ocurre estrictamente en C++, sin generar penalizaciones en el rendimiento. La Figura 2.11 muestra la arquitectura de `py2PowerDEVS` integrada

en el ecosistema de PowerDEVS. Una nueva Capa de Abstracción en Python (PAL) permite acceder desde Python al motor de simulación de PowerDEVS y a los modelos. La vinculación entre Python y PowerDEVS se implementa utilizando la biblioteca Boost.Python [6] y también se enlaza con las bibliotecas compartidas de PowerDEVS.

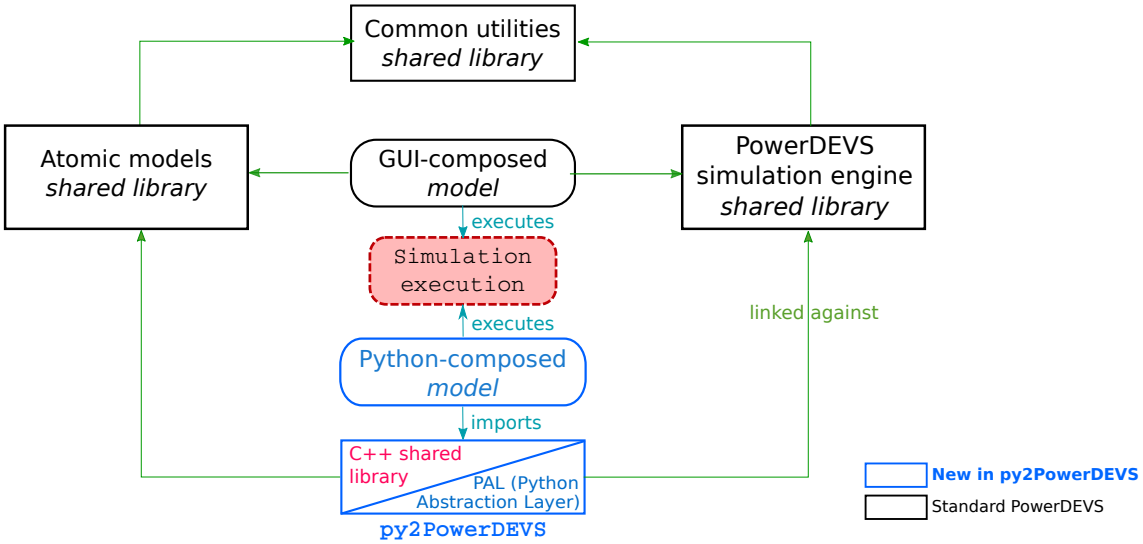


Figura 2.11: Arquitectura y componentes de `py2PowerDEVS`. Borde negro: PowerDEVS estándar. Borde azul: Nuevo framework de scripting `py2PowerDEVS`.

En tiempo de compilación, la PAL genera automáticamente una clase accesible desde Python para cada modelo atómico de PowerDEVS definido en C++. Estas clases están disponibles en el módulo `py2powerdevs.core` de Python. Contar con acceso a los modelos atómicos DEVS desde Python permite definir clases que imitan modelos acoplados DEVS. Además, la PAL genera clases en Python para todos los modelos encontrados en las bibliotecas de PowerDEVS, haciendo que los modelos acoplados también estén disponibles en código Python. Al utilizar la PAL, los scripts en Python pueden acceder al motor de simulación de PowerDEVS para ejecutar modelos, establecer parámetros, configurar registros y seleccionar opciones globales disponibles para los modelos compilados en C++.

2.3.2 Detalles sobre la Implementación de PowerDEVS

La Figura 2.12 muestra el diagrama de clases de PowerDEVS (con excepción de `py2PowerDEVS`). Todos los modelos atómicos DEVS en PowerDEVS son cla-

ses derivadas de la clase `Simulator`. De acuerdo con su comportamiento, estas clases implementan su correspondiente definición de estado, las funciones de transición interna (δ_{int}) y externa (δ_{ext}), la función de salida (λ) y la función de avance de tiempo (ta).

Los coordinadores en PowerDEVS pertenecen a la clase `Coupling`. Debido a la propiedad de clausura bajo acoplamiento, esta también es una clase derivada de `Simulator`. La clase `RootCoupling` deriva de `Coupling` y se diferencia en el objeto padre.

Según la definición formal en la Ecuación 2.1, un modelo acoplado DEVS contiene una lista de conexiones internas y externas con modelos hijos atómicos o acoplados, y mantiene una lista ordenada de eventos para dichos modelos. En PowerDEVS, esta lista de eventos ordenados dentro de un objeto `Coupling` se implementa como un heap binario llamado `ChildHeap`, donde el tiempo del evento se almacena en un vector de C++ y el modelo asociado en un arreglo de punteros a objetos `Simulator`.

Las conexiones internas (IC), las conexiones de entrada externas (EIC) y las conexiones de salida externas (EOC) dentro de un objeto `Coupling` se gestionan como arreglos de punteros a objetos de tipo `Connection`. Por lo tanto, cada vez que un evento llega o se propaga, es necesario recorrer las listas correspondientes.

La clase `Connection` está formada por cuatro números enteros que representan los modelos origen y destino, junto con sus respectivos puertos.

La clase `RootSimulator` en PowerDEVS es la encargada de ejecutar la simulación mediante el avance en el tiempo, cumpliendo así el rol de coordinador raíz en el simulador abstracto DEVS.

2.4 Políticas de Scheduling en Colas de Eventos

En la simulación de Sistemas de Eventos Discretos (DES), la gestión eficiente de los eventos futuros resulta esencial para garantizar la correcta evolución del modelo en el tiempo simulado [17]. En este contexto, las *colas de prioridad* juegan un papel determinante, pues permiten organizar la lista de eventos fu-

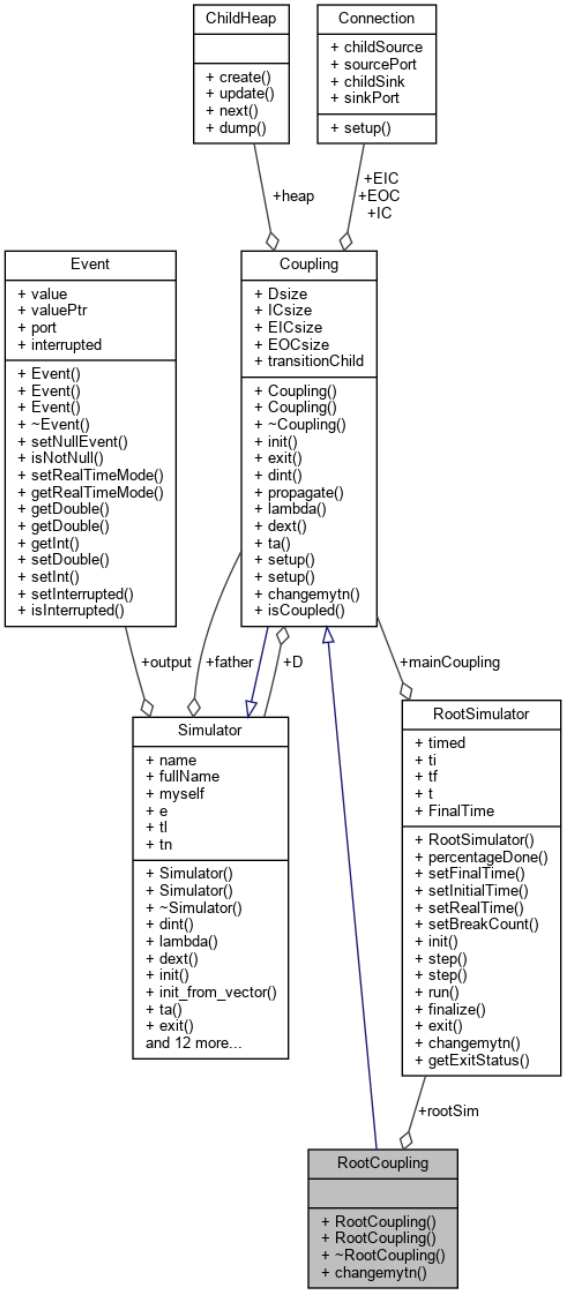


Figura 2.12: Diagrama de Clases del motor de PowerDEVS (clases: `Simulator`, `Coupling`, `Connection`, `RootSimulator`, `Event`, `ChildHeap` y `RootCoupling`).

turos (*Future Event List*, FEL) de acuerdo a las marcas de tiempo (*timestamps*) de sus eventos. De esta manera, se asegura que en cada iteración del simulador se procese el evento cronológicamente más próximo, manteniendo la coherencia temporal de la simulación.

Sin una estructura de datos eficiente, la selección del siguiente evento implicaría recorrer la lista de todos los eventos pendientes, lo cual resultaría computacionalmente costoso, especialmente en modelos complejos con miles o millones de eventos.

Las *colas de prioridad* suelen permitir las siguientes operaciones:

- **Inserción**
- **Extracción del elemento de mayor prioridad**
- **Cambio de prioridad de un elemento**.

A continuación se presentarán algunas variantes de colas de prioridad utilizadas más frecuentemente: *Heap Binario* (Sección 2.4.1), *Paring Heap* (Sección 2.4.2) y *Splay Tree* (Sección 2.4.3). Como se mencionó en la sección anterior, PowerDEVS utiliza un *Heap Binario* para manejar el scheduling de eventos dentro de la clase `ChildHeap`. No obstante, la utilización de *Pairing Heap* o *Splay Tree* podrían conllevar un manejo más eficiente de los eventos como se analizará más adelante.

2.4.1 Heap Binario

El *heap binario* es una estructura de datos fundamental en la informática, ampliamente utilizada para implementar colas de prioridad, algoritmos de planificación (*scheduling*), ordenamientos (*heapsort*) y sistemas de administración de recursos. Se trata de un árbol binario casi completo que satisface una propiedad específica de orden: en un *max-heap* cada nodo padre tiene un valor mayor o igual que sus hijos, mientras que en un *min-heap* cada nodo padre tiene un valor menor o igual que sus hijos.

Formalmente, un *heap binario* es un árbol binario casi completo en el cual todos los niveles, salvo posiblemente el último, están completamente llenos y los

nodos se ubican lo más a la izquierda posible. Esta propiedad permite representar la estructura de forma compacta en un arreglo unidimensional.

En la práctica, un *heap binario* se almacena eficientemente en un arreglo. Dado un nodo ubicado en la posición i, su hijo izquierdo se encuentra en la posición $2i$, su hijo derecho en la posición $2i + 1$ y su padre en la posición $\lfloor i/2 \rfloor$. Esta representación evita el uso de punteros explícitos y optimiza la manipulación de nodos durante las operaciones de inserción y extracción. La Figura 2.13 muestra un ejemplo de un *min-heap* binario con diez nodos. Obsérvese cómo cada nodo padre tiene un valor menor o igual que sus nodos hijos.

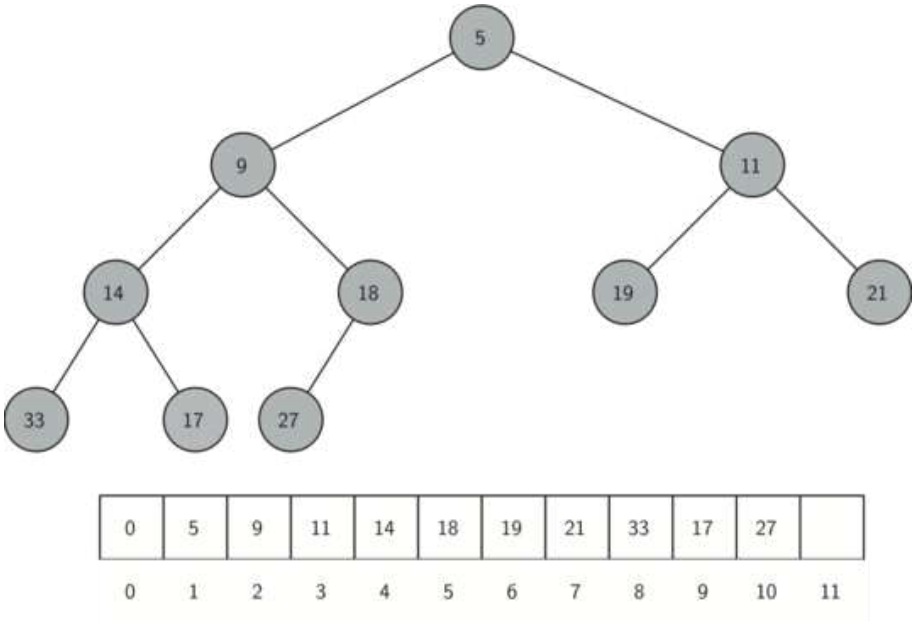


Figura 2.13: Ejemplo de un heap binario representado como árbol (arriba) y su equivalente implementación práctica en un arreglo (abajo).

Las operaciones básicas que ofrece un *heap binario* son:

- **Inserción:** Se añade el nuevo elemento al final del arreglo (manteniendo la completitud del árbol) y luego se reajusta mediante la operación de *sift-up*, que compara el nodo con su padre y lo intercambia si es necesario. La complejidad de esta operación es de $\mathcal{O}(\log n)$ dado que a lo sumo se realizan $\log n$ comparaciones, debido a la altura del árbol.

- **Extracción del mínimo:** Se elimina el elemento raíz (mínimo, en un min-heap), se reemplaza con el último nodo del árbol y se realiza la operación

de *sift-down* para restablecer la propiedad de heap. La complejidad de esta operación es de $\mathcal{O}(\log n)$ por razones análogas a la operación anterior.

- **Cambio de prioridad:** Se modifica el valor del elemento y se realiza la operación de *sift-down* o *sift-up* según corresponda para restablecer la propiedad de heap. También tiene una complejidad de $\mathcal{O}(\log n)$.

2.4.2 Pairing Heap

El *pairing heap* [15] es una estructura de datos autoajustable que pertenece a la familia de los heaps. Fue propuesto como una alternativa práctica a los heaps binomiales y Fibonacci, con el objetivo de simplificar su implementación manteniendo un rendimiento competitivo en operaciones clave como la inserción, la eliminación del mínimo y la disminución de clave.

Un *pairing heap* se organiza como un árbol multi-hijo que preserva la propiedad de heap: el valor de cada nodo es menor o igual que el de sus hijos (en el caso de un *min-heap*). Su funcionamiento se basa en operaciones de fusión (*meld*), mediante las cuales dos heaps se combinan en uno solo comparando sus raíces y anidando el heap de mayor raíz como sub-árbol del menor. La Figura 2.14 ilustra la estructura de un *pairing heap* y muestra la operación de fusión.

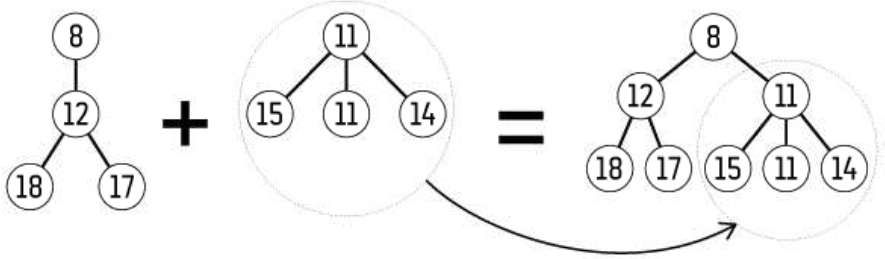


Figura 2.14: Ejemplo de estructura de un *pairing heap* (izquierda) y el proceso de fusión de dos *pairing heaps* (derecha). Se observa cómo se comparan las raíces y se enlaza el heap con mayor raíz como subárbol del otro.

Las operaciones principales del *pairing heap* incluyen:

- **Inserción:** Se interpreta como la fusión de un heap existente con un heap nuevo compuesto por el nodo a insertar. Su complejidad en el peor caso es $\mathcal{O}(1)$.

- **Eliminación del mínimo:** Se elimina la raíz, fusionando de forma recursiva sus sub-árboles hijos mediante una estrategia de emparejamiento de a pares, de donde surge su nombre. La complejidad amortizada de esta operación es de $\mathcal{O}(\log n)$. Por complejidad amortizada, se entiende que si bien la complejidad del peor caso para una operación puede ser de $\mathcal{O}(n)$, para un conjunto de n operaciones, el costo promedio por operación se mantiene en $\mathcal{O}(\log n)$.

- **Cambio de prioridad:** Se corta el sub-árbol que tiene al nodo en cuestión como raiz y se fusiona con la raíz. Se cree que la complejidad amortizada de esta operación es de $\mathcal{O}(\log n)$, pero todavía no ha podido ser demostrado formalmente [14].

Gracias a su simplicidad y buen comportamiento práctico, el *pairing heap* es una opción atractiva para aplicaciones donde se requiere una cola de prioridad eficiente, como simuladores de eventos discretos de gran escala. En tales escenarios, la operación de *cambio de prioridad* es especialmente relevante para reordenar eventos cuando se actualizan sus timestamps durante la simulación.

2.4.3 Splay Tree

El *Splay Tree* es una estructura de datos autoajustable que pertenece a la familia de los árboles de búsqueda binaria. Fue introducido por Sleator y Tarjan en [30] con el objetivo de optimizar el acceso a elementos mediante una estrategia de reorganización dinámica. Su principio fundamental radica en la operación de *splaying*, que mueve el nodo accedido a la raíz del árbol mediante una secuencia de rotaciones. Esta estrategia favorece el acceso rápido a elementos que se utilizan con mayor frecuencia, logrando un tiempo de acceso amortizado logarítmico. En la Figura 2.15 se puede ver cómo el proceso de mover un nodo hacia la raíz (splaying) puede ayudar a mantener el balance del árbol.

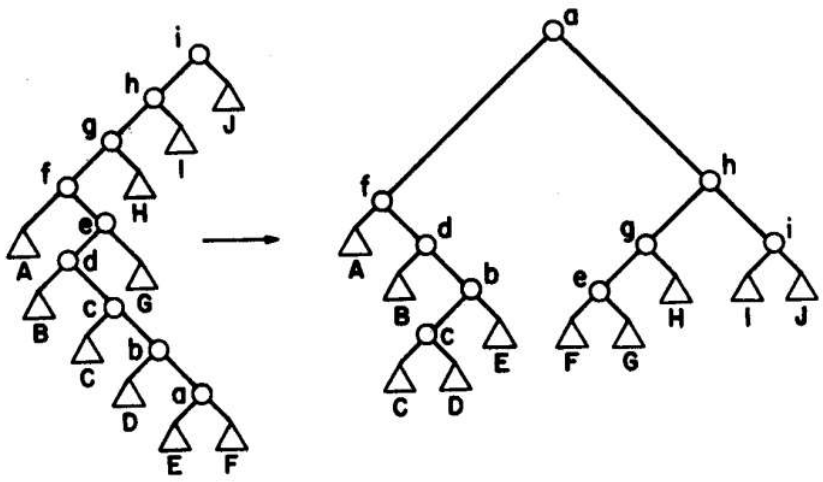


Figura 2.15: Ejemplo de estructura de un *Splay Tree*. El árbol de la derecha muestra el resultado del proceso de *splaying* sobre el nodo de valor **a**.

En el contexto de colas de prioridad, un *Splay Tree* puede emplearse para almacenar eventos con marcas de tiempo. El nodo con menor clave —es decir, el evento más próximo en el tiempo— siempre se localiza accediendo iterativamente al nodo más a la izquierda del árbol. Tras cada extracción, la operación de *splaying* reorganiza el árbol para mantenerlo equilibrado de forma adaptativa, sin necesidad de estructuras adicionales ni balances explícitos.

Las operaciones relevantes al usar un *Splay Tree* como cola de prioridad son:

- **Inserción:** Se inserta el nuevo nodo siguiendo las reglas de un árbol binario de búsqueda. Luego, se aplica la operación de *splaying* para mover el nuevo nodo a la raíz, mejorando su accesibilidad futura.

- **Eliminación del mínimo:** Se localiza el nodo más a la izquierda (el de menor clave), que representa el elemento de mayor prioridad (menor clave). Tras eliminarlo, se aplica *splaying* al nodo padre.

- **Cambio de prioridad:** Se elimina el nodo en cuestión como en un árbol binario común, se inserta un nuevo nodo con la prioridad nueva y luego se aplica *splaying* sobre dicho nodo.

La complejidad amortizada de estas tres operaciones es de $\mathcal{O}(\log n)$ [30].

En resumen, el *Splay Tree* combina la simplicidad estructural de un árbol de búsqueda binaria con la autoajustabilidad necesaria para mantener acotada la

altura del mismo, ofreciendo una alternativa viable a otras colas de prioridad más rígidas. Además, su uso como cola de prioridad resulta atractivo cuando existe una fuerte localidad de referencia.

2.5 Medición de Rendimiento de una Simulación DEVS

En esta tesis definimos **rendimiento** como el *tiempo necesario para ejecutar una simulación de principio a fin*. Para realizar un análisis comparativo del rendimiento de PowerDEVS antes y después de introducir cambios, tuvimos que desarrollar un banco de pruebas. Para esto vamos a valernos de DEVStone [16].

2.5.1 Benchmark de desempeño DEVStone

Dentro de la comunidad DEVS, existe un benchmark reconocido como estándar *de-facto* para evaluar y comparar el rendimiento de simuladores: el benchmark **DEVStone** [16]. Este benchmark ha sido revisado y ampliado en trabajos más recientes [27], donde además se utilizó para realizar comparaciones de performance entre cinco simuladores DEVS ampliamente difundidos. Adicionalmente, DEVStone se emplea habitualmente como herramienta para validar mejoras de rendimiento en simuladores específicos, tal como se reporta en estudios recientes [23, 7].

El benchmark DEVStone define un conjunto de modelos DEVS *sintéticos* con estructuras, tamaños y grados de conectividad variables. Un modelo DEVStone se basa en un modelo acoplado DEVS que replica de forma recursiva una estructura regular, organizada en diferentes niveles de anidación. Cada nivel contiene un número configurable de modelos acoplados y atómicos interconectados según distintos patrones de entrada y salida. El submodelo más interno en la recursión representa la unidad más sencilla: un modelo acoplado con un único modelo atómico.

La flexibilidad de DEVStone se logra mediante cinco parámetros clave: (a) el **Tipo** del modelo define la estructura de conectividad, (b) el **Ancho** (w) indica el

número de componentes atómicos por nivel, (c) la **Profundidad** (d) determina la cantidad de niveles anidados, y (d) y (e) los **Retardos de transición interna/externa** especifican el tiempo que cada función de transición consume. Para ello, se utiliza código específico (Dhrystone benchmark [34]) que garantiza que cada transición interna o externa consuma exactamente el tiempo configurado. De este modo, se logra emular escenarios de carga de CPU realistas dentro de la simulación.

Durante la simulación, todos los modelos atómicos tienen un puerto de entrada y uno de salida. Al activarse, su función de salida genera un mensaje con un único valor entero, independientemente del contenido de su puerto de entrada. Este diseño permite mantener el consumo de memoria bajo control y evitar acumulaciones excesivas de mensajes en la estructura jerárquica.

Para iniciar la simulación, se inyecta un estímulo inicial en los puertos de entrada del modelo acoplado superior. El tiempo total de simulación se mide desde la introducción de este estímulo hasta que no quedan eventos pendientes por procesar.

A continuación se describen los tres **Tipos** clásicos de modelos DEVStone implementados en este trabajo como parte de la validación de mejoras de rendimiento en PowerDEVS: los modelos **LI** (Low Interconnectivity), **HI** (High Interconnectivity) y **HO** (High Output). Estos tipos permiten explorar diferentes patrones de conectividad y carga de eventos en simulaciones de referencia. Para cada uno se presentan ecuaciones que permiten caracterizar su estructura y comportamiento. A partir de los parámetros definidos es posible calcular la cantidad de modelos atómicos (N_{ATOMIC}), la cantidad de eventos generados (N_{EVENTS}), las conexiones de entrada externas (N_{EIC}), las conexiones de salida externas (N_{EOC}) y las conexiones internas (N_{IC}).

2.5.2 Modelos LI (*Low Interconnectivity*)

Los modelos **LI** constituyen la variante más simple de DEVStone, y sirven como línea base para analizar el impacto mínimo de interconexiones. Cada nivel de profundidad contiene un modelo acoplado y $w - 1$ modelos atómicos. El

nivel más interno se limita a un único modelo atómico, tal como se ilustra en la Figura 2.16.

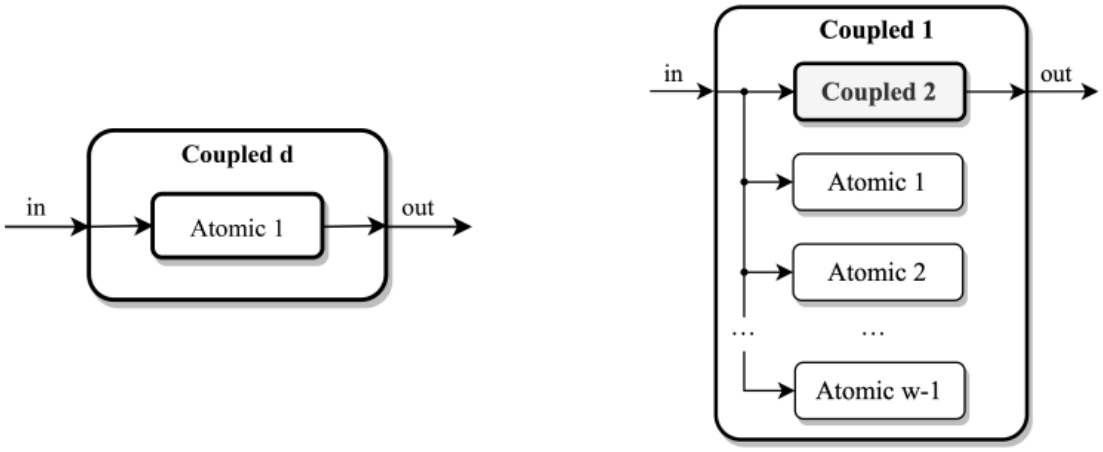


Figura 2.16: Estructura del modelo DEVStone tipo LI (tomado de [7]): (a) submodelo base y (b) modelo acoplado superior.

En estos modelos, los acoplamientos se limitan a conectar el puerto de entrada del nivel superior con los componentes internos (atómicos y acoplados) y a dirigir la salida del modelo acoplado interno hacia el puerto de salida del nivel superior. No se definen acoplamientos internos adicionales, de modo que la estructura resulta lineal y de baja complejidad de mensajes. En las siguientes ecuaciones se describen sus características principales.

$$\begin{aligned} N_{\text{ATOMIC}} &= (w-1) \times (d-1) + 1 \\ N_{\text{EVENTS}} &= w \times (d-1) + 1 \\ N_{\text{EIC}} &= w \times (d-1) + 1 \\ N_{\text{EOC}} &= d \\ N_{\text{IC}} &= 0 \end{aligned} \tag{2.7}$$

2.5.3 Modelos HI (*High Interconnectivity*)

Los modelos **HI** extienden la definición de los LI añadiendo un mayor grado de interconexión interna entre los modelos atómicos de cada nivel. Como se muestra en la Figura 2.17, se agregan conexiones internas (IC) que encadenan

cada par de modelos atómicos adyacentes (en el mismo nivel), incrementando así la complejidad de la secuencia de eventos generados.

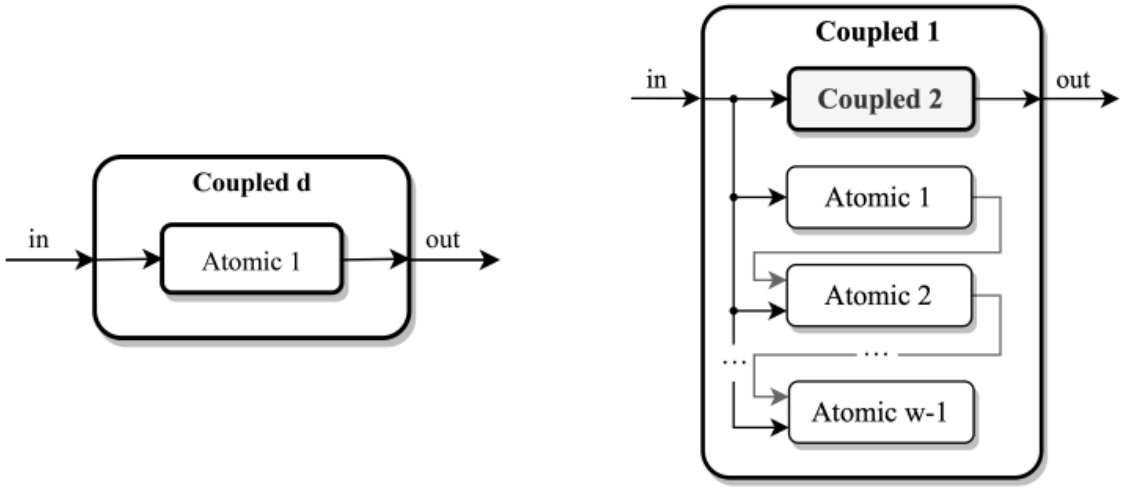


Figura 2.17: Estructura del modelo DEVStone tipo HI (tomado de [7]): (a) submodelo base y (b) modelo acoplado superior.

La estructura de capas y la cantidad de componentes permanecen sin cambios respecto de los LI, pero las conexiones internas generan una mayor cantidad de eventos y una topología que ejerce más presión sobre la gestión de listas de eventos y sincronización en el motor de simulación. Este patrón es útil para evaluar cómo se comporta el simulador bajo condiciones de conectividad más densa. En las siguientes ecuaciones se pueden ver sus características principales.

$$
\begin{aligned}
N_{\text{ATOMIC}} &= (w-1) \times (d-1) + 1 \\
N_{\text{EVENTS}} &= 1 + (d-1) \times \frac{w(w+1)}{2} \\
N_{\text{EIC}} &= w \times (d-1) + 1 \\
N_{\text{EOC}} &= d \\
N_{\text{IC}} &= \begin{cases} (w-2) \times (d-1), & \text{si } w > 2 \\ 0, & \text{si } w \leq 2 \end{cases}
\end{aligned}
\tag{2.8}
$$

2.5.4 Modelos HO (*High Output*)

Finalmente, los modelos **HO** amplían aún más la complejidad de los HI al incorporar puertos de entrada y salida adicionales. Como se aprecia en la Figura 2.18, cada modelo acoplado en la jerarquía tiene ahora dos puertos de entrada y dos de salida. Además, cada modelo atómico dispone de una conexión de salida adicional, lo que multiplica la cantidad de caminos por donde se propagan los mensajes.

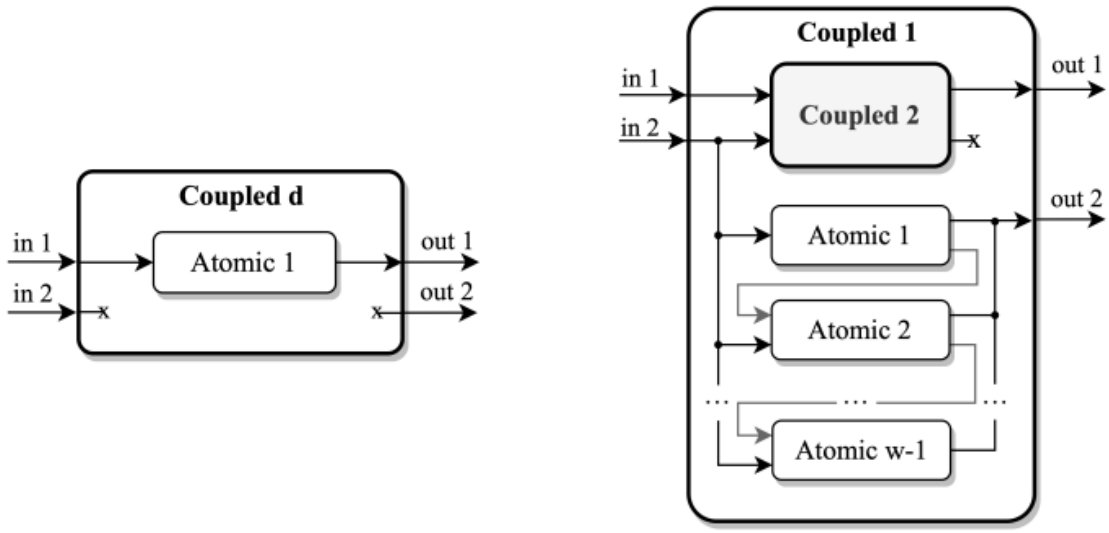


Figura 2.18: Estructura del modelo DEVStone tipo HO (tomado de [7]): (a) submodelo base y (b) modelo acoplado superior.

Este diseño incrementa no solo el tráfico de eventos dentro de la jerarquía, sino que también permite detectar problemas de limpieza de memoria en simuladores que no gestionen correctamente eventos en puertos no conectados. La estructura HO representa así el caso de prueba más exigente dentro de los patrones DEVStone básicos, sirviendo como escenario de validación para analizar la robustez de optimizaciones como el achatamiento de jerarquías o la optimización de algoritmos de encolado. En las siguientes ecuaciones se pueden ver sus características principales.

$$
\begin{aligned}
N_{\text{ATOMIC}} &= (w-1)\times(d-1)+1 \\
N_{\text{EVENTS}} &= 1+(d-1)\times\frac{w(w+1)}{2} \\
N_{\text{EIC}} &= (w+1)\times(d-1)+1 \\
N_{\text{EOC}} &= w\times(d-1)+1 \\
N_{\text{IC}} &= \begin{cases} (w-2)\times(d-1), & \text{si } w>2 \\ 0, & \text{si } w\leq 2 \end{cases}
\end{aligned}
\tag{2.9}
$$

Parte II

Contribuciones Originales

3

Comparación del Rendimiento y Resultados en Simulaciones de Eventos Discretos

En este capítulo se presenta el entorno de trabajo desarrollado como parte de esta Tesis para realizar la comparación automática del rendimiento y de los resultados de la simulación en PowerDEVS. En la Sección 3.1 detallaremos el desarrollo de una herramienta que hace uso de herramientas estándar de *profiling* para la detección de posibles puntos de mejora del rendimiento de PowerDEVS. Estas herramientas pueden ser usadas tanto para PowerDEVS como para cualquier otra herramienta de simulación basada en C++. Por lo tanto, este entorno de trabajo podría fácilmente adaptarse para su utilización con otros simuladores de eventos discretos. En las Secciones 3.2.1 y 3.2.3 se presentan las herramientas para verificar que los resultados de las ejecuciones de distintas versiones PowerDEVS sean equivalentes. Estas versiones corresponderán a la introducción de las mejoras que se analizarán en el Capítulo 4.

3.1 Herramientas de *Profiling*

Comenzamos desarrollando un entorno de trabajo para el profiling de la simulación de modelos utilizando un conjunto de herramientas estándar de debugging y profiling reconocidas en la industria, como *Valgrind*, *Callgrind* y

Massif. Los datos obtenidos, junto con las visualizaciones generadas por *KCachegrind* y *Massif-Visualizer*, permiten producir gráficos informativos que muestran la distribución del uso de recursos computacionales (CPU y memoria) durante la simulación.

A continuación se analizarán los resultados obtenidos con las distintas herramientas de profiling tomando como ejemplo la ejecución de una simulación en PowerDEVS de un modelo básico de redes de datos. En la Figura 3.1 se muestra un gráfico generado por *Massif-Visualizer* a partir de la información generada por *Massif*. En él se puede apreciar el consumo de memoria en el heap según clases a lo largo de la ejecución de la simulación. En la Figura 3.2a se muestra la salida generada con *KCacheGrind* a partir de la información generada por *Callgrind*. En él se pueden apreciar las funciones ejecutadas a lo largo de la simulación ordenadas por su impacto en el tiempo de ejecución total. Por último, en la Figura 3.2b se muestra un árbol de llamadas generado con *KCacheGrind* a partir de la información generada por *Callgrind*.

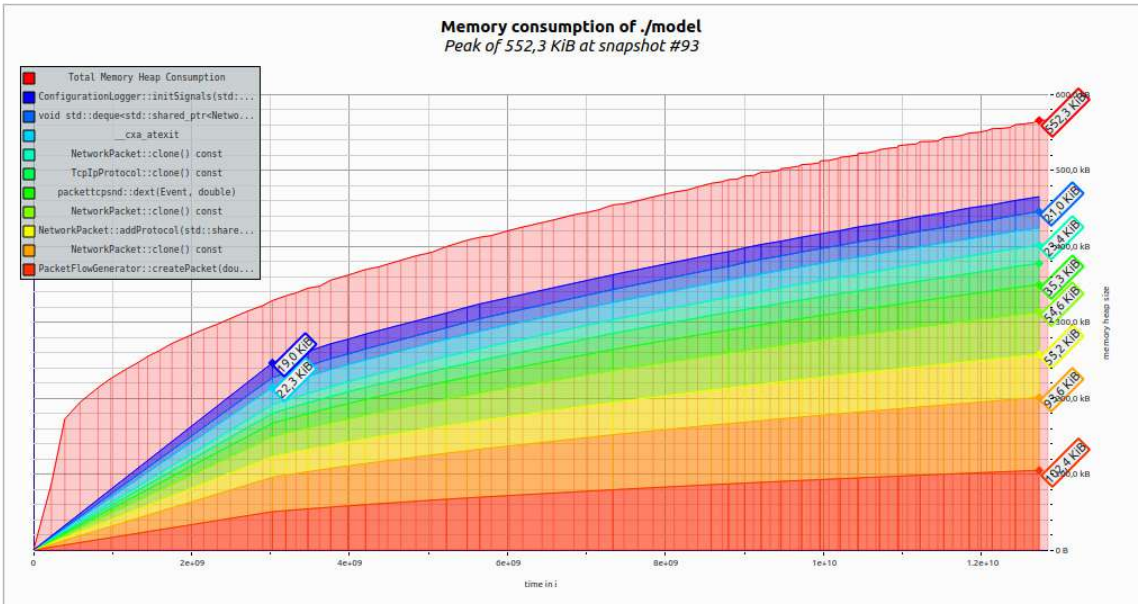
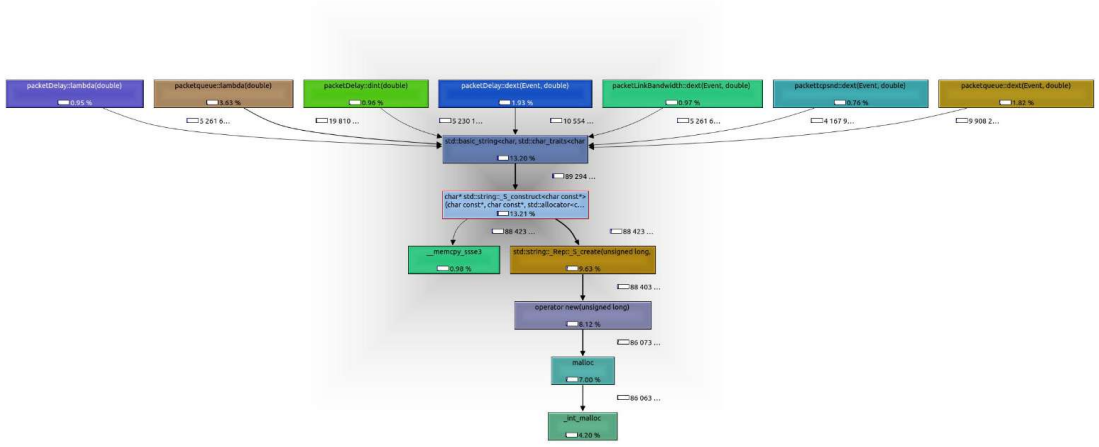


Figura 3.1: Salida de *Massif-Visualizer*.

Cabe destacar que estas herramientas también son válidas para el análisis de modelos DEVS construidos con `py2PowerDEVS` (Sección 2.11), ya que Python se utiliza únicamente para definir la estructura del modelo, mientras que el código ejecutable está completamente implementado en C++.

Incl.	Self	Called	Function	Location
9.01	9.01	583 660 745	DEBUG(DebugEvent, Si…	libengine.so: pdevslib.cpp
8.53	8.53	120 432 451	ChildHeap::update(unsi…	libengine.so: coupling.cpp, stl_vector.h
6.84	6.83	115 284 578	_int_free	libc-2.17.so
67.23	6.28	31 377 741	<cycle 3>	libengine.so
5.63	5.60	115 284 596	_int_malloc	libc-2.17.so
37.32	4.36	59 522 991	Coupling::lambda(doubl…	libengine.so: coupling.cpp, shared_ptr_base.h, …
4.21	4.21	235 057 864	__memcmp_sse4_1	libc-2.17.so
12.42	3.93	88 410 026	ConfigurationLogger::lo…	libmodels.so: ConfigurationLogger.cpp, stl_tre…
17.40	3.79	31 377 741	<cycle 4>	libengine.so
9.38	3.75	115 284 536	malloc	libc-2.17.so

(a)



(b)

Figura 3.2: (a) Salida de *KCacheGrind*. (b) Árbol de llamadas generado con *KCacheGrind*.

3.1.1 Generación Automática de Gráficos de Comparación de Consumo de Recursos Computacionales

Con el objetivo de poder comparar fácilmente el consumo de recursos de cómputo a través de distintas versiones del mismo software, se desarrolló una herramienta en Python que compila, ejecuta y compara los tiempos de ejecución en ambas versiones. Asumiendo que cada versión del software corresponde a un commit en un repositorio de git, para compararlas se debe pasar como parámetros: (a) los números de *hash* de los commits de git a comparar, y (b) el comando necesario para ejecutar la simulación. A continuación se muestra un ejemplo genérico que compara dos versiones de `<software>` en dos commits `<commit_A>` y `<commit_B>`:

```
profilingComparativo.py <commit_A> <commit_B> <software>
```

Como resultado, la herramienta genera un gráfico como el que se puede observar en la Figura 3.3. En él se muestran los tiempos de ejecución (expresados en segundos) por función para cada versión del mismo software. Las barras corresponden a las distintas versiones (i.e. commits) del software.

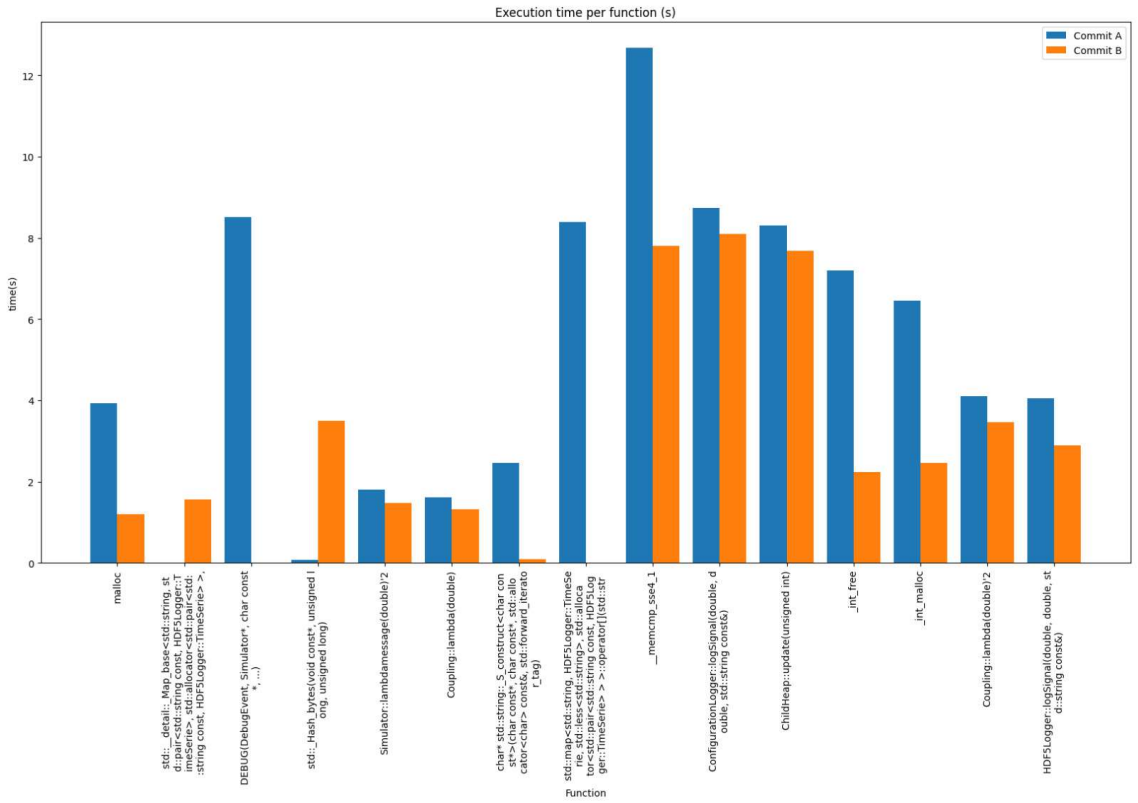


Figura 3.3: Gráfico comparativo de tiempos de ejecución (eje Y) por función (eje X) para dos versiones de un mismo software de simulación (en este caso PowerDEVS).

Para generar este gráfico, el script se sitúa sobre el primer commit a analizar (*Commit A*), compila el software (en este caso PowerDEVS) y a continuación lo ejecuta (i.e. ejecuta la simulación) con $N = 6$ veces. En la primera ejecución utiliza la herramienta *Callgrind*, mientras que las ejecuciones restantes no la utilizan para obtener un tiempo promedio de ejecución total de la simulación. Luego, se repite el mismo procedimiento para el segundo commit a comparar (*Commit B*). Por último, se post-procesa la información generada por *Callgrind* para obtener los porcentajes del tiempo de ejecución total que insumió cada función en cada

versión, y utilizando el tiempo total de ejecución de cada simulación, se calculan los tiempos de cómputo de cada función en particular. Toda esta información se utiliza para generar la Figura 3.3 resultante.

3.2 Adaptación de la Herramienta para PowerDEVS

La herramienta presentada en la sección anterior es genérica y podría utilizarse para comparar distintas versiones de cualquier software basado en C++. No obstante, a continuación analizaremos su utilización para comparar distintas versiones de PowerDEVS. Estas versiones corresponderán a la introducción de distintos cambios (detallados en el Capítulo 4) con el foco puesto en reducir el tiempo consumido en la ejecución de una simulación (i.e. mejora del rendimiento).

3.2.1 Comparación Automática de Resultados de Simulaciones de Modelos

Con el objetivo de verificar que los cambios introducidos no afecten la correctitud de las simulaciones, se desarrolló un script en `Python` que permite comparar automáticamente las series temporales almacenadas en los archivos de log generados por los modelos, ya sea entre distintas versiones del simulador PowerDEVS o entre diferentes versiones de modelos atómicos de propósito general.

En los casos en que ambas simulaciones producen series temporales registradas en los mismos instantes de tiempo, se calcula directamente el *error cuadrático medio normalizado por el rango dinámico* (DRMSE), definido como:

$$\text{DRMSE} = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(v_1^{(i)} - v_2^{(i)}\right)^2}}{\text{maxVal} - \text{minVal}} \tag{3.1}$$

donde v_1 y v_2 son vectores de longitud n que representan los valores de las dos series a comparar, y maxVal y minVal corresponden al valor máximo y mínimo considerando ambas series. Esta métrica produce un valor adimensional que

permite cuantificar el error relativo, independientemente de la escala absoluta de los datos.

En cambio, si las series no comparten los mismos instantes de registro, se realiza una interpolación cruzada utilizando *splines* cúbicos: se interpola la serie v_1 en los instantes de t_2 y, recíprocamente, la serie v_2 en los instantes de t_1. De esta forma se obtienen dos nuevas series temporales evaluadas en un conjunto común de puntos de tiempo, lo que permite aplicar la métrica definida en la Ecuación 3.1. Para dicha interpolación se emplea la función `interp1d` del paquete `SciPy`. Esta interpolación y comparación se realiza por medio de la función `compareTimeSeriesDifferentTimes()`, detallada en el Código 3.1.

Código 3.1: Función para comparar series temporales con instantes de muestreo distintos.

```
from scipy.interpolate import interp1d

def compareTimeSeriesDifferentTime(t1, v1, t2, v2,
    ↪ interpolation='cubic'):
    t_common = np.union1d(t1, t2)
    posx1_interp = interp1d(t1, v1, kind=interpolation,
    ↪ fill_value="extrapolate")(t_common)
    posx2_interp = interp1d(t2, v2, kind=interpolation,
    ↪ fill_value="extrapolate")(t_common)

    maxVal = max(np.max(v1), np.max(v2))
    minVal = min(np.min(v1), np.min(v2))
    dynamicRange = maxVal - minVal

    if dynamicRange == 0:
        return 0.0
    else:
        rmse = math.sqrt(mean_squared_error(posx1_interp,
    ↪ posx2_interp))
        return rmse / dynamicRange
```

En esta función, t_1 y v_1 representan los instantes de tiempo y valores de la primera serie temporal, respectivamente, mientras que t_2 y v_2 corresponden a los de la segunda serie. La función devuelve la raíz del error cuadrático medio relativo, normalizada al rango dinámico global de ambas señales originales.

3.2.2 Instrumentación del Motor del Simulador Abstracto de PowerDEVS

Independientemente de la comparación de los resultados de la simulación antes y después de la introducción de cambios llevada a cabo en la Sección 3.2.1, también se deben comparar los mensajes intercambiados internamente dentro del motor del simulador PowerDEVS.

Con el fin de abordar el análisis del rendimiento de manera rigurosa, se dotó a PowerDEVS de un nuevo mecanismo de registro de datos (*data-logging*) en el núcleo del simulador. El mecanismo original de logging de eventos en el motor de PowerDEVS (opción `-d`) era ineficiente (ver Sección 4.2.1) y no era útil para la comparación automática de distintas ejecuciones de una simulación. Nuestro nuevo sistema de registro es independiente del registro que se realiza a nivel de modelo, ya que no se encarga de los datos producidos por los modelos en sí (como en la Sección 3.2.1) sino del funcionamiento y la mensajería interna del simulador.

Se re-instrumentaron las clases `Simulator`, `Coupling`, `RootSimulator` y `RootCoupling` (Sección 2.3.2) de forma tal que, cada vez que se envía un mensaje o se propaga un evento, se registran una o dos líneas en un archivo de log en formato CSV denominado `pdevsDebug.log`. Este archivo contiene información relevante, como el nombre completo del objeto, el tipo de evento, el tiempo virtual de simulación y el tiempo real de ejecución. A continuación se muestra cómo luce una línea genérica de este archivo:

```
<real_time>,<simu_time>,<type>,<event>,<fullname>,<src_name>,
<src_port>,<sink_name>,<sink_port>,<e>,<ta>
```

El nombre completo del objeto (campo `<fullname>`) incluye los nombres de todos sus objetos padres, separados por puntos. El nivel de verbosidad del regis-

tro puede configurarse mediante un nivel de depuración (`engine_debug_level`), como se explica en la Sección 4.2.1. El mecanismo de logging original, que imprimía mensajes de texto por línea de comandos tales como: ''`<nombre atomico>, Execution Init Function at <tiempo>`'',''`<nombre atomico>, Execution Output Function at <tiempo>`'',''`<nombre atomico>, Execution Internal Transition at <tiempo>`'', etc., resultaba ser demasiado verboso para la comparación automática de los mismos, y aportaba una menor cantidad de información por evento.

Se desarrollaron algoritmos para comparar sistemáticamente los archivos de log generados en dos ejecuciones distintas de un mismo modelo. Para cada línea en uno de los archivos, se identifica la línea correspondiente en el otro archivo utilizando el nombre completo del objeto involucrado.

Una vez aplicadas las optimizaciones que se describirán en el Capítulo 4 y tras acelerar exitosamente la ejecución de la simulación, es necesario garantizar que los resultados sean correctos. Esto implica que tanto los resultados de la simulación (producidos por el modelo) como los mensajes internos intercambiados en el núcleo del simulador antes y después de la introducción de los cambios deben ser equivalentes. Por *equivalente* se entiende que, para modelos que no modifican su estructura durante la simulación, *los mensajes internos deben ser* ***exactamente los mismos***.

En las siguientes secciones se utilizarán estas nuevas capacidades de registro y las herramientas desarrolladas para realizar comparaciones y *verificar la correctitud* de las mejoras implementadas. El sistema de registro fue habilitado únicamente al comparar la correctitud de las trazas de eventos (antes y después de las mejoras), y se mantuvo deshabilitado al comparar el rendimiento de la simulación, con el fin de eliminar la sobrecarga introducida por el logging.

3.2.3 Comparación Automática de Logs del Motor de Simulación

Aprovechando el mecanismo de logging de mensajes del motor de simulación de PowerDEVS descrito en la sección anterior, se desarrolló una herramienta

en Python para verificar la correctitud de las mejoras implementadas comparando los logs generados previa y posteriormente a las mismas. Esto significa que los mensajes internos intercambiados en el núcleo del simulador antes y después de la introducción de los cambios deben ser equivalentes.

Teniendo en cuenta que se dotó a PowerDEVS con la funcionalidad de aplanamiento automático de modelos (que se describirá más adelante en la Sección 4.2.2), se optó por comparar solamente los eventos generados por modelos atómicos. Se considera que los cambios mantuvieron la *correctitud* si, al comparar los logs correspondientes a las ejecuciones antes y después de su introducción, *los eventos producidos por los modelos atómicos se encuentran en exactamente el mismo orden*. Esta comparación se realiza por medio de la función `compareEngineLogs()`, detallada en el Código 3.2. La función `filterDfToCompare(log)` se encarga de filtrar el dataframe del log, dejando solo aquellas filas correspondientes a modelos atómicos. Cabe destacar que para lidiar con logs de gran tamaño (que en algunos casos de interés del Capítulo 5 llegan a ocupar hasta 9 GB), estos son procesados de a fragmentos.

Código 3.2: Función para comparar series temporales con instantes de muestreo distintos.

```
def compareEngineLogs(logA, logB):
  df_sample = pd.read_csv(logA,nrows=1000)
  df_sample_size = df_sample.memory_usage(index=True).sum()
  my_chunk = (1000000000 / df_sample_size)/1000
  my_chunk = int(my_chunk//1)# create the iterator

  logA_iter = pd.read_csv(logA, usecols=columns, dtype=dtypes,
   ↪ iterator=True, chunksize=my_chunk)
  logB_iter = pd.read_csv(logB, usecols=columns, dtype=dtypes,
   ↪ iterator=True, chunksize=my_chunk)

  logA = logA_iter.get_chunk()
  logA_stepIndexes = logA[logA["event"]=="STEP"]
  while len(logA_stepIndexes) == 0:
    logA = logA_iter.get_chunk()
    logA_stepIndexes = logA[logA["event"]=="STEP"]
```

```
15
16  logA_atomics = logA.loc[logA_stepIndexes.index[0]:]
17  logB = logB_iter.get_chunk()
18  logB_stepIndexes = logB[logB["event"]=="STEP"]
19  while len(logB_stepIndexes) == 0:
20    logB = logB_iter.get_chunk()
21    logB_stepIndexes = logB[logB["event"]=="STEP"]
22  logB_atomics = logB.loc[logB_stepIndexes.index[0]:]
23  logA_atomics = filterDfToCompare(logA)
24  logB_atomics = filterDfToCompare(logB)
25
26  are_Same = True
27  logA_index = 0
28  logB_index = 0
29  while True:
30    while logA_index >= len(logA_atomics):
31      try:
32        logA = logA_iter.get_chunk()
33        logA_atomics = filterDfToCompare(logA)
34        logA_index = 0
35      except StopIteration:
36        #If the end of log is reached, both files must end
37          if logB_index != len(logB_atomics):
38            return False
39          else:
40            return True
41    while logB_index >= len(logB_atomics):
42      try:
43        logB = logB_iter.get_chunk()
44        logB_atomics = filterDfToCompare(logB)
45        logB_index = 0
46      except StopIteration:
47        #If the end of log is reached, both files must end
48        if logA_index != len(logA_atomics):
49          return False
50        else:
51          return True
52      if logA_atomics.iloc[logA_index].equals(logB_atomics.iloc[
```

```
    ↪ logB_index]):
53          logA_index += 1
54          logB_index += 1
55        else:
56          are_Same = False
57          return are_Same
```

3.2.4 Generación Automática de Métricas de Simulación

Con las herramientas descritas hasta el momento, es posible obtener diversas métricas relevantes sobre el rendimiento de la simulación, así como información detallada acerca de la cantidad y el tipo de eventos procesados por cada modelo, ya sea atómico o acoplado.

En las Figuras 3.4 y 3.5 se muestran algunas de las métricas obtenidas a partir del archivo `pdevsDebug.log`, generado por una simulación del modelo de la red de adquisición de datos de ATLAS, descrito en la Sección 5.2. Para este experimento, se habilitó el registro detallado de la mensajería interna del simulador abstracto.

Los gráficos permiten observar la cantidad de transiciones internas y externas, así como el tiempo total de ejecución acumulado para cada tipo de transición, desglosado por modelo atómico. Esta información resulta especialmente útil para identificar cuellos de botella o comportamientos anómalos en la dinámica del modelo simulado.

Una métrica clave utilizada para evaluar las mejoras de rendimiento obtenidas a partir de optimizaciones en el simulador es la *aceleración relativa* (*relative speedup*), definida como:

$$RS = (et_{bc} - et_{ac}) \,/ et_{bc} \cdot 100, \tag{3.2}$$

donde RS representa la aceleración relativa expresada en porcentaje, et denota el tiempo de ejecución (medido en tiempo real de reloj de pared), y los subíndices bc y ac indican los tiempos *antes* (*before changes*) y *después* (*after changes*) de aplicar las modificaciones, respectivamente.

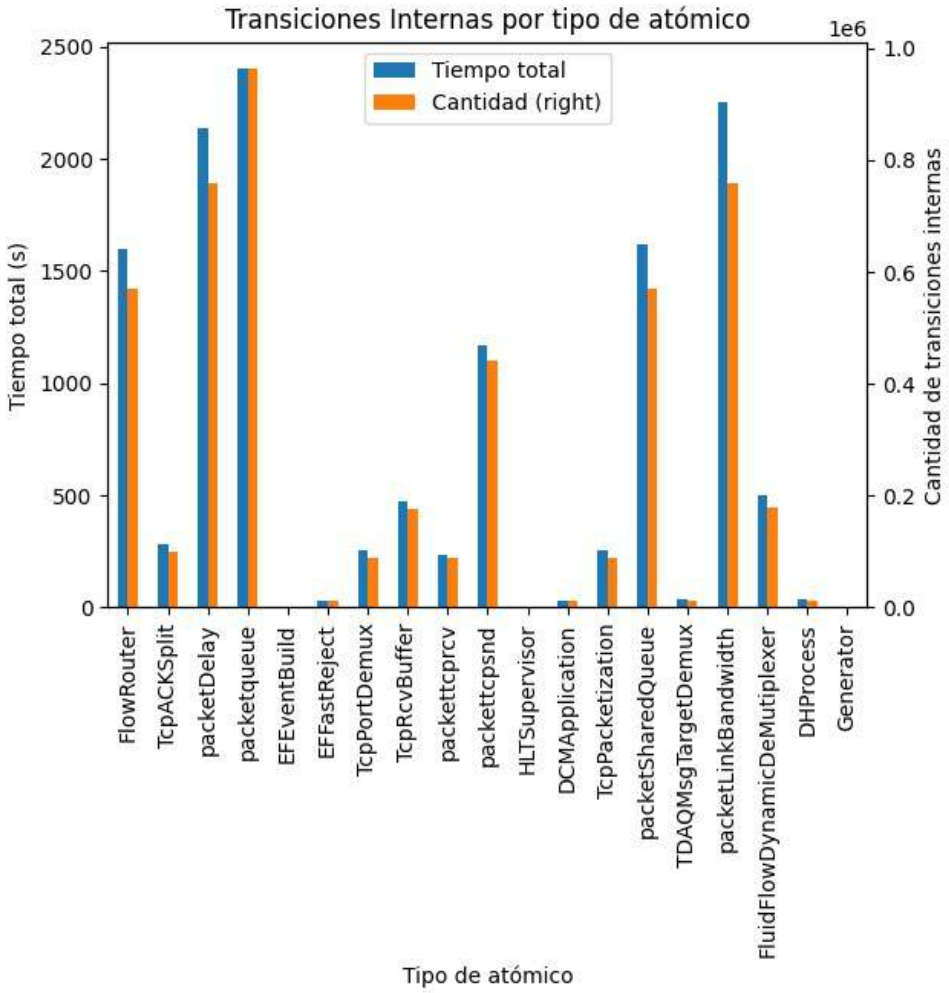


Figura 3.4: Cantidad de transiciones internas y tiempo total de ejecución de las mismas, agregado por modelo atómico, en una simulación del modelo TDAQ.

Un valor positivo de *RS* indica una mejora en el rendimiento de la simulación posterior a los cambios (analizados en el Capítulo 4). Por el contrario, un valor de 0 % implica que no hubo variación en el tiempo de ejecución, lo cual equivale a un factor de aceleración igual a uno.

3.2.5 Implementación de Herramientas de Integración Continua

Con el fin de asegurar una base de comparación para las subsiguientes modificaciones que se realicen en PowerDEVS, se seleccionó una batería de modelos (de entre los que se distribuyen junto con PowerDEVS) que cubren distintos dominios (modelos de sistemas continuos, discretos e híbridos), aplicaciones

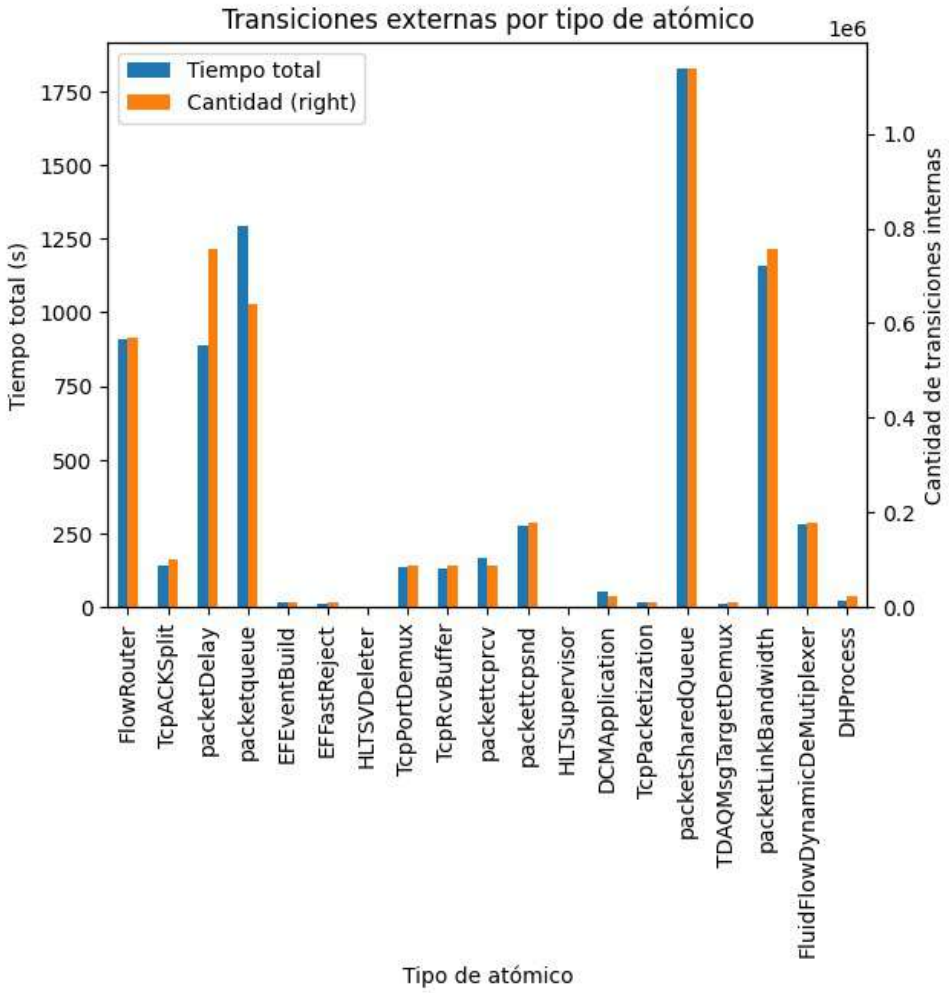


Figura 3.5: Cantidad de transiciones externas y tiempo total de ejecución de las mismas, agregado por modelo atómico, en una simulación del modelo TDAQ.

(e.g. modelos biológicos, epidemiológicos, eléctricos, mecánicos) y definiciones (e.g. py2PowerDEVS y VECDEVS). Estos modelos, junto con las herramientas de comparación desarrolladas en las Secciones 3.2.1 y 3.2.3, serán utilizados para realizar ensayos de regresión.

3.2.5.1 Batería de Modelos para Ensayos de Regresión

Los ejemplos utilizados fueron agrupados de acuerdo con la naturaleza de los modelos. En primer lugar, se consideraron modelos de tiempo continuo definidos por conjuntos de ecuaciones diferenciales ordinarias (ODEs), tales como: el clásico modelo presa-depredador de Lotka-Volterra (`lotka volterra`), un modelo no lineal rígido (`stiff`), y un modelo concentrado de una línea de transmisión

sin pérdidas compuesto por una red de 50 circuitos tanque LC (`transmission line`) implementado utilizando VECDEVS [3].

En segundo lugar, se abordaron tres ejemplos representativos de modelos definidos por ecuaciones diferenciales con retardo (DDEs), incluyendo una red neuronal celular (`cellular network spikes`) y dos modelos adicionales ampliamente citados en la literatura (`oberle and pesch` y `hairer et al`) [10].

Para los modelos de tiempo discreto, se incluyeron: un oscilador de segundo orden construido mediante la conexión en serie de dos retardos (`q-operator`), y el conocido modelo de Nicholson-Bailey para la evolución de dos poblaciones (`nicholson bailey`), el cual guarda una estrecha relación con el modelo de Lotka-Volterra pero en el dominio discreto.

En el caso de los modelos estocásticos, se utilizó una red de datos simple (`network basic`) que incluye un generador de paquetes, un receptor, un enrutador y un modelo de colas (`queueing`) con generación de trabajos, encolamiento y procesamiento de tareas.

Finalmente, se incluyeron varios ejemplos de sistemas dinámicos híbridos, tales como: una pelota que rebota descendiendo por una escalera (`Bouncing Ball`), un convertidor Buck conmutado DC-DC (`Buck Controlled Coupled`), un motor de corriente continua (`DC Drive` y `DC Drive Buck`), una neurona de tipo espiga (`Spiking Neuron`) y un péndulo invertido en lazo cerrado (`Inverted Pendulum`).

Esta batería de modelos será utilizada en el Capítulo 5 para validar los cambios introducidos en el Capítulo 4.

4

Mejoras Introducidas en el Motor de PowerDEVS y en Atómicos de Propósito General

Debido a la separación estricta entre los modelos y el simulador abstracto del formalismo DEVS, se pueden analizar las mejoras en cada uno de ellos de forma separada. Dado que el simulador abstracto es común a todos los modelos, una mejora en este tendrá impacto, con distinto grado de mejora, en la simulación de todos los modelos. Por otro lado, las mejoras introducidas en los modelos atómicos de propósito general utilizados en la construcción de muchos otros modelos más complejos también tendrán impacto, aunque probablemente de menor magnitud y de forma menos general.

4.1 Detección de oportunidades de mejora de desempeño en PowerDEVS

En este capítulo se ilustrará de qué manera fue aplicado el framework presentado en el Capítulo 3 para identificar cuellos de botella en el rendimiento y áreas con potencial de mejora dentro de la simulación. En el contexto de este trabajo de Tesis, evaluamos el **rendimiento de una simulación** en términos del

tiempo total de ejecución requerido para completar una simulación, es decir, el tiempo punta a punta (*end-to-end*) desde la inicialización del modelo hasta la finalización del proceso de simulación.

Se llevó a cabo un análisis sistemático de las causas que originaban pérdidas de rendimiento, así como de los cambios implementados en PowerDEVS con el objetivo de mitigarlas. En las siguientes secciones se detallan dichas modificaciones, organizadas en dos grandes categorías: (i) cambios introducidos en el núcleo del simulador (Sección 4.2) y (ii) mejoras aplicadas a los modelos atómicos de propósito general (Sección 4.3). Asimismo, se pudieron detectar y corregir múltiples fugas de memoria (*memory leaks*) que afectaban negativamente la eficiencia y estabilidad del sistema durante simulaciones muy prolongadas y/o con una alta carga de eventos.

4.2 Mejoras en el Motor de PowerDEVS

4.2.1 Mejoras en el Sistema de Debugging Preexistente

Una contribución en la mejora del rendimiento y la reducción del tiempo total de simulación se logró mediante la sustitución de las funciones tradicionales de *debug logging* del motor del simulador (Código 4.1) por una macro optimizada (Código 4.2). Esta modificación permite evitar el costo asociado a las llamadas a funciones en tiempo de ejecución, ya que las macros se expanden directamente en el código durante el proceso de compilación, eliminando así la sobrecarga funcional y mejorando la eficiencia general del simulador. Esto permite que, cuando está deshabilitado, no afecte significativamente el rendimiento de la simulación.

Código 4.1: Función de logging del motor de PowerDEVS.

```
1 void DEBUG(DebugEvent de,Simulator *s, const char *fmt, ...)
2 {
3     if (!debug) return;
4     va_list listPtr;
5     va_start( listPtr, fmt );
```

```
    vprintf( fmt, listPtr );
    va_end( listPtr );
}
```

Código 4.2: Nueva macro de logging del motor de PowerDEVS.

```
#define DEBUG(__rt, __st, __ds, __de, ...) \
    if(engineDebugLevel) \
        _DEBUG(__rt, __st, __ds, __de, __VA_ARGS__);
```

Paralelamente, se perfeccionó el mecanismo de recopilación y visualización de la información de depuración (como se mencionó en la Sección 3.2.2), permitiendo una trazabilidad más precisa y eficiente de los eventos internos del simulador. Estas mejoras resultan especialmente útiles durante las etapas de validación y verificación del comportamiento del modelo, ya que permiten detectar discrepancias o comportamientos no deseados entre distintas ejecuciones.

El mecanismo de logging original (por línea de comandos) se habilitaba con la opción `-d` mientras que el nuevo mecanismo se habilita con la opción `-engineDebugLevel <arg>`. El argumento `<arg>` sigue la sintaxis `ABCDEF.G`, donde cada carácter representa una opción distinta de configuración.

El primer carácter, `A`, indica qué tipo de mensajes de modelos deben ser registrados. Puede tomar los siguientes valores: `0` para registrar únicamente mensajes de modelos atómicos, `1` para registrar solo mensajes de modelos acoplados, o `2` para registrar mensajes de ambos tipos de modelos.

Los cinco caracteres siguientes, `BCDEF`, controlan la activación del registro para distintos tipos de mensajes del motor de simulación: `B` activa el registro de mensajes δ_{int}, `C` de mensajes λ, `D` de mensajes ta, `E` de mensajes δ_{ext}y `F` de mensajes `propagate`. Cada uno de estos puede habilitarse o deshabilitarse de forma independiente, donde `1` indica que se habilita el registro y `0` que se desactiva.

El último carácter, `G`, es opcional y especifica la proporción del tiempo de simulación durante la cual se activará el registro de mensajes del motor. En caso de omitirse, el mecanismo estará activo durante el total del tiempo de simulación. Este período está siempre centrado respecto al tiempo total de simulación, lo que significa que se registra un mismo porcentaje de tiempo al inicio y al

final de la ejecución. Esta funcionalidad fue incorporada para hacer frente al gran volumen de mensajes que puede generar el motor en algunos modelos, lo que haría impráctico registrar todos los mensajes durante toda la duración de la simulación.

Por ejemplo, el parámetro `-engineDebugLevel 110011.1` indica que se deben registrar los mensajes δ_{int}, δ_{ext}y `propagate` exclusivamente para modelos acoplados. Además, estos mensajes se registrarán únicamente durante el primer 5 % y el último 5 % del tiempo total de simulación.

En conjunto, estas modificaciones contribuyen a acelerar la ejecución de la simulación (i.e. mejorar el rendimiento) de manera significativa cuando el logging está deshabilitado, al tiempo que preservan la capacidad de inspección detallada cuando está habilitado.

4.2.2 Aplanamiento Automático de Modelos

Otro aspecto en el que se observó que se podía mejorar el rendimiento de una simulación es la simplificación de la estructura del modelo. Se incorporó la funcionalidad opcional de *aplanamiento automático de modelos*, cuyo propósito es mejorar la eficiencia general, simplificando su estructura mediante una reducción significativa en la complejidad del mecanismo de mensajería interna utilizado por el simulador.

En el contexto de un modelo totalmente aplanado, es importante destacar que existe un único modelo DEVS acoplado (*top model*) que opera junto con un coordinador, el cual asume además las responsabilidades del coordinador raíz. Esto da lugar a la creación de una única cola de eventos dentro del marco del modelo. En consecuencia, la política específica que se implemente para la gestión de dicha cola de eventos en esta configuración aplanada se espera que tenga una influencia significativa sobre el rendimiento y el comportamiento general de la simulación (ver Sección 4.2.4).

Dado que el proceso de aplanamiento debe ser accesible para modelos PowerDEVS desarrollados tanto mediante la interfaz gráfica de usuario (GUI) como a través de la extensión `py2PowerDEVS`, el aplanamiento se realiza en *tiempo de*

ejecución (no mediante un preprocesamiento), durante la fase inicial de la inicialización del modelo. Esto contrasta con el mecanismo desarrollado en [1], el cual se ejecuta antes de correr la simulación pero solo es válido para modelos creados con la GUI.

La activación del aplanamiento de modelos se lleva a cabo mediante el uso del parámetro `-flatten <arg>` en la línea de comandos, donde el argumento define la máxima cantidad de niveles permitidos (el valor por omisión es 0).

4.2.2.1 Implementación del Mecanismo de Aplanamiento Automático

La rutina `flattenRecursion` (1) implementa una estrategia de aplanamiento jerárquico que permite transformar modelos DEVS con múltiples niveles de acoplamientos en una estructura sin niveles compuesta exclusivamente por modelos atómicos. En esta implementación, la variable `startingFrom` representa el nivel de profundidad de la jerarquía desde el que se inicia el proceso de aplanamiento.

El algoritmo comienza recorriendo todos los hijos del modelo acoplado actual (líneas 4 a 14 pseudocódigo del 1). Para cada hijo, evalúa su tipo:

- Si el hijo es un modelo acoplado, la función se invoca recursivamente sobre él. Como resultado, se obtiene su lista de hijos aplanados, los cuales se integran en la lista temporal del modelo superior.

- Si el hijo es un modelo atómico, se agrega directamente a la lista temporal.

Una vez procesados todos los hijos, se reemplazan las conexiones internas del modelo acoplado en cuestión (líneas 15 a 31). Esta sustitución es fundamental para garantizar que todas las referencias a submodelos internos se traduzcan a conexiones planas, válidas en la nueva estructura sin jerarquía.

Posteriormente, se actualizan las conexiones externas de salida (*outputs*) (líneas 31 a 38). En el caso de las conexiones externas de salida provenientes de un modelo hijo acoplado M_i por su puerto p_i hacia el puerto de salida p_j, se agrega una nueva conexión externa de salida para cada hijo del modelo M_i que tuviera una conexión externa de salida al puerto p_i.

Algorithm 1 Pseudocódigo de la función `flattenRecursion`

```
 1: procedure FLATTENRECURSION(startingFrom, level)
 2:     Inicializar lista temporal de hijos aplanados.
 3:     if level >= startingFrom then
 4:         for all hijos M_i del modelo acoplado do
 5:             if M_i es de tipo Coupling then
 6:                 M_i.flattenRecursion(startingFrom, level +1).
 7:                 Agregar hijos de M_i a la lista temporal.
 8:                 for all (src, src_port, dst, dst_port) ∈ M_i.IC do
 9:                     Creo una conexión interna (src, src_port, dst, dst_port).
10:                 end for
11:             else
12:                 Agregar M_i a la lista temporal.
13:             end if
14:         end for
15:         for all (src, src_port, dst, dst_port) ∈ IC do
16:             if src y dst son ambos de tipo Coupling then
17:                 for all modelo atómico hijo m_i de src | (m_i, p_i, 0, src_port) ∈ src.EOC do
18:                     for all modelo atómico hijo m_j de dst | (0, dst_port, m_j, p_j) ∈ dst.EIC do
19:                         Creo una conexión interna (m_i, p_i, m_j, p_j).
20:                     end for
21:                 end for
22:             else if Sólo src es de tipo Coupling then
23:                 for all modelo atómico hijo m_i de src | (m_i, p_i, 0, src_port) ∈ src.EOC do
24:                     Creo una conexión interna (m_i, p_i, dst, dst_port)
25:                 end for
26:             else if Sólo dst es de tipo Coupling then
27:                 for all modelo atómico hijo m_j de dst|(0, dst_port, m_j, p_j) ∈ dst.EIC do
28:                     Creo una conexión interna (src, src_port, m_j, p_j).
29:                 end for
30:             end if
31:         end for
32:         for all (src, src_port, 0, dst_port) ∈ EOC do
33:             if src es de tipo Coupling then
34:                 for all modelo atómico hijo m_i de src | (m_i, p_i, 0, src_port) ∈ src.EOC do
35:                     Creo una conexión externa de salida (m_i, p_i, 0, dst_port)
36:                 end for
37:             end if
38:         end for
39:         for all (0, src_port, dst, dst_port) ∈ EIC do
40:             if dst es de tipo Coupling then
41:                 for all modelo atómico hijo m_j de dst|(0, dst_port, m_j, p_j) ∈ dst.EIC do
42:                     Creo una conexión externa de entrada (0, src_port, m_j, p_j).
43:                 end for
44:             end if
45:         end for
46:         Agrego todos los modelos de la lista temporal a la lista de hijos del acoplado
47:     end if
48: end procedure
```

De igual modo, se reemplazan las conexiones externas de entrada (*inputs*) (líneas 39 a 45), asegurando que todos los eventos entrantes se enruten de forma coherente hacia los puertos correctos de los modelos atómicos.

Como resultado, en caso de que el acoplado en cuestión se encuentre a una profundidad mayor o igual a `startingFrom`, se obtiene una lista de modelos hijos que serán todos atómicos, junto con sus conexiones internas, externas de salida y externas de entrada que representan el mismo modelo acoplado que antes, pero ahora sin profundidad.

Cabe aclarar que al aplanar los modelos acoplados hijos, los índices dentro de la lista de modelos cambian debido a que se agregan tantos modelos atómicos como hijos tenga dicho acoplado. Para que los índices en las conexiones sean coherentes, se mantiene una lista con el offset generado por el incremento en acumulado en la cantidad de hijos hasta el momento. Se omitió este detalle en el pseudocódigo 1 por simplicidad.

Este mecanismo de aplanamiento optimiza el rendimiento del simulador *PowerDEVS* ya que reduce la complejidad estructural, facilita el enrutamiento de eventos y mejora la eficiencia del planificador de simulación.

Por otro lado, al aplanar un modelo se observa un incremento en la cantidad total de conexiones dentro del mismo. Esto se debe a que, como se puede ver en los ciclos de las líneas 23, 27, 34 y 41 del pseudocódigo 1, cuando se procesa una conexión correspondiente a un hijo acoplado se debe agregar una conexión por cada modelo que esté conectado a dicho puerto dentro del modelo hijo. Esto se ve exacerbado en caso de que la conexión siendo procesada implique a dos modelos acoplados hijos, como se puede ver en la línea 16 del pseudocódigo 1, ya que debo generar una conexión entre cada modelo atómico conectado al puerto de salida en el modelo hijo fuente con cada uno de los modelos atómicos conectados al puerto de entrada en el modelo hijo destino.

En síntesis, esta rutina permite conservar la modularidad durante el modelado, sin comprometer la eficiencia durante la ejecución de simulaciones de sistemas complejos, con la contrapartida de generar un mayor consumo de memoria para el almacenamiento de las conexiones dentro del modelo. Como se mencionó anteriormente, la métrica de rendimiento de la simulación es el tiempo que toma la ejecución de una simulación de punta a punta. No obstante, esta métrica podría extenderse para incluir también el uso de memoria. Como trabajo a futuro se explorará este aspecto para alcanzar un balance entre el uso de

recursos computacionales.

4.2.3 Mejoras en el Manejo de Conexiones Entre Modelos

Como mencionamos en la sección anterior, el aplanamiento de modelo conlleva un incremento en la cantidad de conexiones totales, además de concentrarlas todas dentro de un único modelo acoplado. Por este motivo, se decidió optimizar las estructuras internas en las que se mantienen los índices correspondientes a las conexiones pertinentes a cada modelo hijo.

Además de los arreglos de punteros *External Input Coupling* (EIC), *External Output Coupling* (EOC) e *Internal Coupling* (IC) utilizados para almacenar las conexiones en la clase `Coupling`, se contaba con *IC_Index*, un vector de vectores de enteros en el que se guardaban los índices en *IC* de las conexiones salientes del i-ésimo modelo hijo (Código 4.3).

Código 4.3: IC_Index en la clase `coupling` antes de las modificaciones.

```
1 //IC_index: for each model, stores outgoing internal
    ↪ connection
2 //index in IC
3 std::vector<std::vector<unsigned int> > IC_Index;
```

Al no contar con las conexiones indexadas por puerto de salida además de por modelo hijo, al momento de propagar un mensaje se debían recorrer todas las conexiones salientes de un modelo hijo buscando aquellas en las que coincida el puerto de salida. Para el caso de las conexiones externas de salida (EOC), al no contar con ninguna estructura para organizar los índices de las conexiones según modelo, se debían recorrer todas las conexiones externas de salida del modelo acoplado buscando aquellas que coincidan en modelo fuente y puerto de salida del mismo (Código 4.4).

Código 4.4: Propagación interna de mensajes previo a los cambios.

```
1 void Coupling::propagate(Event x, Time t){
2   output=x;
3   Port p=output.port;
4   unsigned int i;
```

```
  if(x.isNotNull()){
    for(i=0;i<IC_Index[transitionChild].size();i++){
      int ic=IC_Index[transitionChild][i];
      if (IC[ic]->sourcePort==p){
        output.port=IC[ic]->sinkPort;
        childs[IC[ic]->childSink]->dextmessage(output, t);
        heap.update(IC[ic]->childSink);
      };
    };
    for(i=0;i<EOCsize;i++){
      if (EOC[i]->childSource==transitionChild &&
          EOC[i]->sourcePort==p){
            output.port=EOC[i]->sinkPort;
            father->propagate(output, t);
      };
    };
  };
}
```

Similarmente, para la propagación de mensajes de entrada también se debían recorrer todas las conexiones externas de entrada (EIC) buscando aquellas que correspondan al puerto de entrada pertinente (Código 4.5).

Código 4.5: Propagación de mensajes externos previo a los cambios.

```
void Coupling::dext(Event x, Time t){
  Port port=x.port;
  for(unsigned int i=0;i<EICsize;i++){
    if (EIC[i]->sourcePort==port){
      childs[EIC[i]->childSink]->dextmessage(x, t);
      heap.update(EIC[i]->childSink);
    };
  };
};
```

Por este motivo se decidió modificar el campo *IC_Index* en la clase `coupling`

(Código 4.6) para que esté indexado por modelo y puerto en lugar de solo por modelo, y se agregó una nueva estructura de datos *EOC_Index* para organizar la información de los índices de las conexiones de salida de forma similar. Además se agregó una tercera estructura *EIC_Index* para organizar los índices de las conexiones de entrada según su puerto. Cabe aclarar que todas estas estructuras se populan al iniciar el modelo, haciendo una pasada por *IC*, *EOC* y *EIC*.

Código 4.6: Nuevos campos en la clase `coupling`.

```
//IC_Index: for each child model and port, stores outgoing
   ↪ internal
//connection index in IC
std::vector<std::vector<std::vector<unsigned int>>> IC_Index;
//EOC_Index: for each child model and port, stores outgoing
   ↪ external
//output connection index in EOC
std::vector<std::vector<std::vector<unsigned int>>> EOC_Index;
//EIC_Index: for each external input port, stores exteral
//input connection index in EIC
std::vector<std::vector<unsigned int> > EIC_Index;
```

Esta modificación no solo tuvo un impacto positivo en el rendimiento, al permitir una búsqueda y acceso más eficiente a las conexiones durante la propagación de eventos, sino que también contribuyó a simplificar significativamente la estructura del código fuente. Como resultado, se mejoró la legibilidad del código y se facilitó su mantenimiento y extensión (Código 4.7), aspectos fundamentales en el desarrollo de herramientas de simulación de propósito general como PowerDEVS.

Código 4.7: Nueva versión de los métodos `propagate` y `dext` en la clase `coupling`.

```
void Coupling::propagate(Event x, Time t){
  output=x;
  Port p=output.port;
  unsigned int i;
  if(x.isNotNull()){
```

```
    if(p < IC_Index[transitionChild].size()){
      for(i=0;i<IC_Index[transitionChild][(int)p].size();i++){
        int ic=IC_Index[transitionChild][(int) p][i];
        output.port=IC[ic]->sinkPort;
        childs[IC[ic]->childSink]->dextmessage(output, t);
        heap.update(IC[ic]->childSink);
      }
    }
    if(p<EOC_Index[transitionChild].size()){
      for (i=0; i<EOC_Index[transitionChild][(int) p].size();i
   ↪ ++){
        int eoc=EOC_Index[transitionChild][(int) p][i];
        output.port=EOC[eoc]->sinkPort;
        father->propagate(output, t);
      }
    }
  }
}

void Coupling::dext(Event x, Time t){
  Port port=x.port;
  if(port>=EIC_Index.size()) return;
  for (unsigned int i=0; i<EIC_Index[port].size();i++){
    int eic=EIC_Index[port][i];
    x.port=EIC[eic]->sinkPort;
    childs[EIC[eic]->childSink]->dextmessage(x, t);
    heap.update(EIC[eic]->childSink);
  }
};
```

4.2.4 Cambios en el Sistema de Scheduling de Eventos

Como se mencionó en la sección 2.4, la estructura de datos para el manejo del scheduling de eventos es de vital importancia para un simulador de eventos discretos. Al aplanar un modelo y contar con una gran cantidad de atómicos hijos en un único acoplado, la eficiencia de esta única cola de prioridad toma más relevancia aún.

La gestión de la cola de prioridad que maneja el scheduling de eventos en los coordinadores de PowerDEVS se realiza en la clase `ChildHeap` (Figura 2.12 y Código 4.8). Originalmente, PowerDEVS utilizaba una estructura *binary heap* (Sección 2.4.1).

Código 4.8: Cola de prioridad original en PowerDEVS.

```
class ChildHeap {
    public:
      void create(Simulator **, unsigned int);
      void update(unsigned int);
      int next();
      void dump();
    private:
      std::vector<std::vector<ModelTime> > heap;
      Simulator **D;
      unsigned int size;
};
```

Se creó la clase abstracta `IPriorityQueue` (Código 4.9) manteniendo la interfaz de `ChildHeap` (Código 4.8) para facilitar el cambio entre las distintas implementaciones de colas de prioridad desarrolladas: `Pairing Heap`, `Splay Tree` y `Heap Binario`. Se eligieron estas tres implementaciones debido a su amplio reconocimiento como estructuras de datos eficientes para el manejo de scheduling de eventos en simuladores de eventos discretos [17].

Código 4.9: Clase abstracta IPriorityQueue.

```
class IPriorityQueue {
public:
```

```
3   virtual void create(Simulator **, unsigned int, Coupling*)
    ↪ =0;
4   virtual void update(unsigned int)=0;
5   virtual unsigned int next()=0;
6   virtual void dump()=0;
7   virtual ~IPriorityQueue() {};
8 };
```

En el caso del `Heap Binario`, se decidió utilizar la implementación original de PorweDEVS, por lo que el único cambio necesario fue agregar a la clase `ChildHeap` como subclase de `IPriorityQueue`.

`Pairing Heap` también fue de sencillo desarrollo, ya que se optó por utilizar la implementación incluida en la librería Boost de C++ [6], por lo que solo hubo que adaptarla a la interfaz de `IPriorityQueue`.

Por último, el `Splay Tree` [30] fue implementado desde cero como una subclase de `IPriorityQueue`. En su diseño, se incluyó la particularidad de no encolar en la misma estructura aquellos eventos con un tiempo mayor al tiempo final de simulación, a estos se los apuntaba a un nodo dummy.

La elección de la cola de prioridad a ser utilizada se define mediante el parámetro `-priorityQueue` que se pasa por línea de comandos, que puede ser usado con las opciones: `pairingHeap`, `heap` o `splayTree` (siendo esta última la opción por defecto).

Tanto *Pairing Heap*, como *Splay Tree* resultaron ser más eficientes que *Heap Binario*, siendo *Splay Tree* la opción de mejor rendimiento entre las tres.

4.3 Mejoras en Modelos Atómicos de Propósito General

4.3.1 Mejoras en el Modelo Atómico del Integrador QSS3

El método de integración numérica por omisión en PowerDEVS es QSS de orden 3 o QSS3 [8]. En la implementación de un modelo atómico DEVS para QSS3, para determinar el próximo evento interno luego de la ejecución de las

funciones de transición interna (δ_{int}) o externa (δ_{ext}) se requiere del cálculo de raíces cúbicas. Originalmente, el modelo QSS3 en PowerDEVS calculaba este valor de `sigma` utilizando la función `pow(val, 1.0/3)`. Sin embargo, existe una función más eficiente y más precisa para el cálculo de las raíces cúbicas en C++ llamada `cbrt(val)`. Entonces, se reemplazó la función utilizada por el modelo atómico integrador QSS3 para encontrar las raíces de un polinomio de tercer grado. Esta modificación se propuso con el objetivo de mejorar tanto la eficiencia computacional como la precisión numérica del integrador.

Para evaluar el impacto de este cambio, se construyó un modelo que representa el movimiento de una partícula cargada en un campo magnético constante, descrito por las ecuaciones de Lorentz (tomado de [28]). Este modelo involucra un sistema de seis integradores QSS3 y fue seleccionado específicamente debido a que posee una solución analítica conocida, lo cual permite realizar comparaciones directas entre la simulación y la solución exacta. La implementación del nuevo método de resolución de raíces permitió reducir el tiempo de simulación en un 19 %, evidenciando una mejora significativa en el rendimiento. Además, se observó una leve disminución en el error numérico respecto de la solución analítica, lo cual indica que la mejora en la eficiencia no comprometió sino que mejoró la precisión del resultado.

Este resultado sugiere que al optimizar funciones internas clave, como la resolución de polinomios en atómicos QSS, es posible obtener beneficios computacionales relevantes sin afectar negativamente la calidad de las simulaciones. Este cambio resulta particularmente importante en modelos que emplean múltiples integradores QSS3 en paralelo.

4.3.2 Mejoras en los Modelos Atómicos de *DataLoggers*

Para registrar los resultados de una simulación en PowerDEVS se crearon los atómicos `DiscreteSamplerLogger` y `SamplerLogger`. Estos atómicos hacen uso del mecanismo de data-logging genérico introducido en el trabajo doctoral de Matías Bonaventura [4]. Este mecanismo dota a los modelos atómicos de funciones de logging y permite seleccionar mediante el parámetro `-v` (o

-variable_logging_backend) el formato en el que almacenarán estos datos, cuyas opciones son: CSV, HDF5 o Scilab.

Con el objetivo de mejorar el rendimiento y la eficiencia en la determinación de ciertos valores estadísticos derivados de los datos generados por los modelos atómicos que invocan dichos registradores, se introdujeron modificaciones en las estructuras de datos utilizadas por los modelos atómicos registradores de datos DiscreteSamplerLogger y SamplerLogger. En ellos se utilizaba un vector para mantener la ventana de valores sobre los cuales calcular las métricas (Código 4.10).

Código 4.10: Ventana móvil de valores.

```
class SamplerLogger : public IPowerDEVSLogger {
...
private:
    std::vector<double> valuesInWindow;
...
}

void SamplerLogger::addValueToMovingAvg(double value){
 if(this->valuesInWindow.size() >= this->movingAvgWindowSize){
   this->valuesInWindow.erase(this->valuesInWindow.begin());
   ↪ // remove first (pop_front)
 }
 this->valuesInWindow.push_back(value);
}

double SamplerLogger::getMovingAvg(){
  double sum = 0;
  for (auto val : this->valuesInWindow){
    sum += val;
  }
  return sum / this->valuesInWindow.size(); // sum/N
}
```

Esta se reemplazó por una *double ended queue* o *deque* en la librería estándar de C++. Esta es mucho más eficiente ya que cuenta con la operación `pop_front()` en $\mathcal{O}(1)$ mientras que el remover el primer elemento en un vector toma $\mathcal{O}(n)$. Además, se modificó la forma en la que se calcula la media móvil en ambas clases. Originalmente, cada vez que se debía calcular la media móvil, se recorría la lista de valores en la ventana (función `getMovingAverage()` en Código 4.10), lo que fue reemplazado por mantener una variable con la suma de los valores en la ventana, actualizándola cada vez que se agrega un nuevo valor, y dividiéndola por su tamaño cada vez que sea solicitada la media móvil (Código 4.11).

Código 4.11: Ventana móvil de valores.

```
class SamplerLogger : public IPowerDEVSLogger {
...
private:
    std::deque<double> valuesInWindow;
    double valuesInWindowSum = 0;
...
}

void SamplerLogger::addValueToMovingAvg(double value){
  if(this->valuesInWindow.size() >=
     this->movingAvgWindowSize){
        this->valuesInWindowSum -=this->valuesInWindow.front();
        this->valuesInWindow.pop_front(); // remove first item
    }
    this->valuesInWindowSum += value;
    this->valuesInWindow.push_back(value); // add to back
}

double SamplerLogger::getMovingAvg(){
  if(this->valuesInWindow.size() == 0) return std::nan("");
  return this->valuesInWindowSum/this->valuesInWindow.size();
}
```

Entre las métricas calculadas se incluyen el valor máximo, el mínimo, y el promedio móvil. Estas mejoras permiten un procesamiento más ágil y eficiente de los datos, lo cual es especialmente relevante en simulaciones de larga duración o en aquellas que involucran grandes volúmenes de datos generados a partir de múltiples fuentes.

La reestructuración de estas clases no solo redujo el tiempo de cómputo asociado a las operaciones estadísticas, sino que también facilitó una mejor organización interna del código, contribuyendo a su legibilidad y mantenibilidad.

4.3.3 Mejoras en Modelos Atómicos de Redes de Datos de Propósito General

En el trabajo doctoral de Matías Bonaventura [4] se introdujo la biblioteca de modelos de redes de datos `Network`. Esta biblioteca provee una variedad de modelos atómicos y acoplados de los elementos más comunes en una red de comunicaciones (e.g. routers, nodos, colas, etc.) listos para usar.

Se modificó el mecanismo mediante el cual ciertos modelos atómicos de la biblioteca `Network` gestionan los nombres de las variables o métricas a registrar (con la función `logSignal()`). La implementación original resultaba ser considerablemente costosa en términos de tiempo de ejecución, especialmente en simulaciones de gran escala o con alta frecuencia de registro de eventos. En ella, se llamaba a `logSignal` con strings literales como nombres de señales, lo que hacía que cada vez que se quería añadir un valor al log de dicha señal, se creaba un string y luego se destruía (Código 4.13).

Código 4.12: Logging de variables.

```
1
2 Event PacketFlowGenerator::lambda(double t) {
3     auto packet = createPacket(t);
4
5     this->logger->logSignal(t, this->sigma, "intergen");
6     this->logger->logSignal(t, this->generatedPackets, "count"
  ↪ );
```

```
    this->logger->logSignal(t, packet->getLength_bits(), "
    ↪ sent_bits");
```

Esto fue reemplazado por un conjunto de variables dentro de cada atómico que guardan los nombres de las señales a ser almacenadas en un log (Código 4.13).

Código 4.13: Almacenamiento de nombres de señales de logging en variables.

```
class PacketFlowGenerator: public BaseSimulator {
  ...
private:
  std::string str_forcedGeneration = "forcedGeneration";
  std::string str_intergen = "intergen";
  std::string str_count = "count";
  std::string str_sent_bits = "sent_bits";
  ...
}

Event PacketFlowGenerator::lambda(double t) {
  auto packet = createPacket(t);
  this->logger->logSignal(t, this->sigma, str_intergen);
  this->logger->logSignal(t, this->generatedPackets, str_count
  ↪ );
  this->logger->logSignal(t, packet->getLength_bits(),
  ↪ str_sent_bits);
  ...
}
```

Esta modificación se realizó de manera análoga en todas las señales de logging en los modelos atómicos más frecuentemente utilizados para modelar redes: *PacketFlowGenerator, packetDelay, packetLinkBandwith, packetLogger, packetQueue, packetqueue_simple, packettcpsnd y packetSharedQueue.*

La nueva estrategia adoptada optimiza el acceso y la manipulación de estos nombres, lo que se traduce en una mejora significativa en el rendimiento global

de dichos modelos. Este cambio, aunque localizado, contribuye de manera importante a la eficiencia del sistema en su conjunto, particularmente en entornos donde la captura de datos de logging es intensiva.

Parte III

Casos de Estudio y Resultados

5

Casos de Estudio

En este capítulo se explorará el impacto en el rendimiento de las mejoras presentadas en el Capítulo 4 tanto en el motor del simulador como en algunos modelos atómicos. Estos cambios se validan primero sobre la batería de modelos de la Sección 3.2.5.1. Luego, se aplican sobre dos casos de estudio de interés para el SEDLab que fueron seleccionados porque involucran modelos complejos de gran tamaño y en donde el impacto de las mejoras es más significativo. Finalmente, se aplican sobre el benchmark sintético DEVStone introducido en la Sección 2.5.1.

5.1 Resultados para Batería de Modelos

Inicialmente, se llevó a cabo un estudio de rendimiento sobre los más de 20 modelos de la batería de modelos de la Sección 3.2.5.1 representativos de diversas áreas del conocimiento. El objetivo principal de estas pruebas fue evaluar el impacto de las mejoras introducidas en modelos previamente desarrollados y distribuidos junto con la herramienta PowerDEVS [29]. Asimismo, se buscó analizar si dicho impacto varía dependiendo del tipo de modelo considerado.

A continuación se comparan el rendimiento de la simulación antes y después de aplicar los cambios. Para este último caso, se utiliza *Pairing Heap* y el modelo

aplanado. La Figura 5.1 resume los *speedups relativos* promedios calculados a partir de la Ecuación 3.2. El eje *y* muestra el *speedup relativo* mientras que el eje *x* indica el nombre del modelo de simulación al que corresponde. Los valores han sido ordenados desde los que logran una mayor optimización (izquierda) hacia los que obtienen una menor optimización (derecha). Cada simulación se repitió tres veces, tomando como base un tiempo virtual de simulación relevante para cada modelo, y los resultados fueron promediados. Asimismo, se garantizó que el entorno de hardware donde se realizaron las simulaciones sea controlado y que estas no compitan con otros procesos de usuario relevantes.

Los *speedups relativos* abarcan una amplia gama, desde más del 60 % hasta aproximadamente 0 % (es decir, sin mejora, *speedup* igual a 1), alcanzando incluso un valor negativo en el caso del modelo DEVStone tipo LI indicando que el rendimiento se degradó ligeramente tras aplicar las optimizaciones (cuya explicación se deja para la Sección 5.5).

Se observa que la mayor mejora se logró en el ejemplo de colas de paquetes de datos (`queueing`), desarrollado tanto mediante la interfaz gráfica (GUI) como con `py2PowerDEVS`. Este fue seguido de cerca por los modelos DEVStone y el ejemplo TDAQ (analizados en las Secciones 5.5 y 5.2, respectivamente).

Cabe destacar, además, que en el modelo `network basic` el rendimiento obtenido depende de si el modelo fue construido con la GUI o con `py2PowerDEVS`, dado que, aunque representan el mismo sistema conceptual, sus implementaciones no son exactamente equivalentes.

5.2 Sistema de Adquisición de Datos mediante Redes de Datos del Experimento ATLAS en CERN

5.2.1 Introducción

El *Large Hadron Collider* (LHC) del CERN es actualmente el acelerador de partículas más grande del mundo, con una circunferencia de 27 kilómetros. Su funcionamiento se basa en la colisión de haces de partículas (protones o iones pesados) a una frecuencia extremadamente alta, del orden de una colisión cada

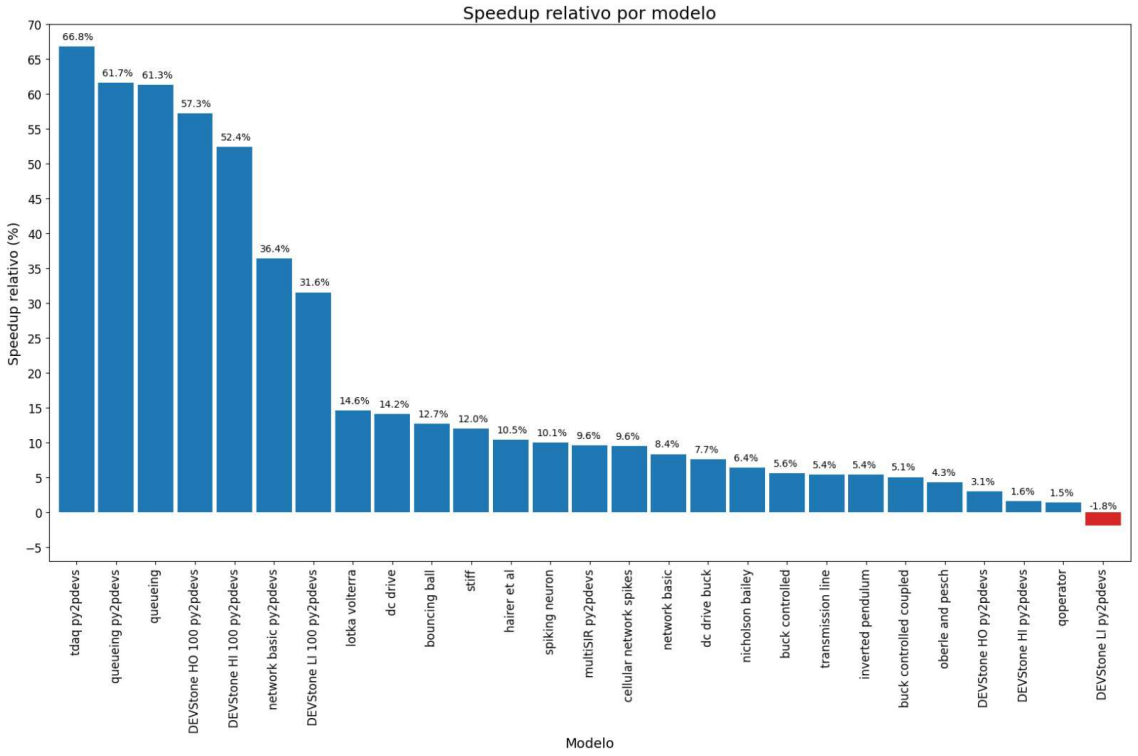


Figura 5.1: Speedups relativos (RS %) promedio para la batería de modelos DEVS en PowerDEVS definida en la Sección 3.2.5.1, abarcando una amplia gama de aplicaciones y construidos tanto con la interfaz gráfica como con `py2PowerDEVS`.

25 ns. Estas colisiones se realizan en las proximidades de detectores de gran escala, como ATLAS, CMS, ALICE y LHCb, entre otros.

Durante el período comprendido entre 2013 y 2015, el LHC suspendió temporalmente sus operaciones para realizar tareas de mantenimiento y actualización, en lo que se denominó *Long Shutdown 1* (LS1). Esta pausa marcó la transición entre la primera campaña de recolección de datos (Run 1) y el reinicio para la segunda fase (Run 2).

En particular, el detector ATLAS tiene la capacidad de registrar eventos de colisión con energías muy elevadas, lo que habilita la exploración de fenómenos físicos de frontera, como la detección del Bosón de Higgs, la búsqueda de dimensiones adicionales o la identificación de posibles componentes de la materia oscura. Cada colisión de haces de partículas da lugar a lo que se denomina un **Evento** (con mayúscula), término usado en física de altas energías (HEP) para referirse a la totalidad de las señales inducidas por partículas que son registradas

y digitalizadas por el detector. Cabe señalar que esta noción difiere de la acepción usada en modelado DEVS, donde se hace referencia a eventos discretos de simulación (en minúsculas).

El volumen de datos crudos generados por el detector ATLAS es del orden de 60 TB/s, lo que representa un desafío sin precedentes para su almacenamiento y procesamiento. Para hacer frente a esta magnitud de información, ATLAS implementa un sistema de filtrado en múltiples capas denominado **Trigger and Data Acquisition** (TDAQ), Figura 5.2. Este sistema decide, en tiempo real, si cada Evento debe almacenarse de manera permanente o puede descartarse sin pérdida de información relevante.

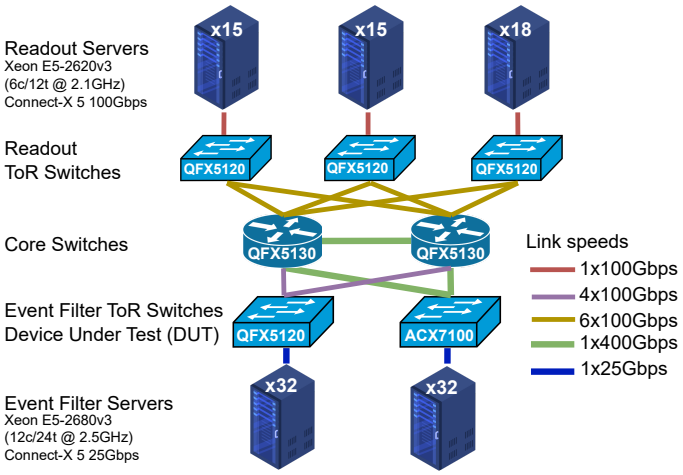


Figura 5.2: Topología del prototipo del sistema de adquisición de datos de TDAQ (tomado de [26]).

El primer nivel de este sistema, llamado **L1 trigger**, realiza una primera reducción de la tasa bruta de eventos, que inicialmente es de 40 millones de Eventos por segundo, filtrando hasta alcanzar una tasa de 100,000 Eventos por segundo. Los Eventos aceptados por L1 son almacenados temporalmente en el sistema de lectura (*Read-Out System*, ROS), en forma de estructuras de datos denominadas *fragmentos*. Estos fragmentos son luego accedidos por una segunda etapa de filtrado, el **High-Level Trigger** (HLT), en el cual algoritmos físicos re-analizan la información con una granularidad distinta. El HLT retiene solamente 1,000 Eventos por segundo considerados *interesantes* desde el punto de vista científico.

Todo este proceso de adquisición, filtrado y análisis de datos se sustenta sobre una infraestructura computacional altamente distribuida, cuya eficiencia y

escalabilidad son claves para el éxito científico del experimento. En particular, la red de datos que interconecta al HLT con el ROS constituye un subsistema crítico cuya complejidad lo convierte en un candidato ideal para ser modelado y analizado mediante técnicas de simulación discreta orientada a eventos. Esta motivación da lugar al siguiente apartado, donde se describe cómo se ha utilizado PowerDEVS para representar y estudiar el comportamiento dinámico del sistema TDAQ.

5.3 Simulación del Sistema TDAQ de ATLAS

Dada la complejidad estructural y dinámica del sistema **Trigger and Data Acquisition** (TDAQ) de ATLAS, se ha utilizado la herramienta PowerDEVS para su modelado y simulación. Este modelo representa una red de adquisición de datos de gran escala, estructurada como un sistema distribuido compuesto por miles de nodos de cómputo interconectados mediante una red Ethernet jerárquica. Para reflejar con fidelidad tanto la lógica funcional como la estructura de comunicación del sistema real, se implementaron más de 8,700 submodelos DEVS interrelacionados [5].

En este escenario, la performance de ejecución del simulador resulta un factor determinante, ya que la viabilidad del modelo como herramienta de apoyo a decisiones depende directamente de la obtención de resultados en tiempos razonables. Un estudio reciente basado en este modelo permitió estimar los requerimientos de hardware para una futura expansión del sistema TDAQ, aportando información valiosa para su diseño y puesta en marcha [26]. Sin embargo, debido a los tiempos de simulación prohibitivamente largos, dicho modelo solo abarcó aproximadamente el 10 % de la red real. Esta restricción evidencia una limitación crítica: sin mejoras sustanciales en el rendimiento del simulador, la capacidad de realizar múltiples simulaciones completas a escala real para comparar el desempeño de distintas configuraciones se vuelve inviable, especialmente considerando que el tamaño del sistema crecerá en los próximos años.

Frente a este desafío, se identificó la necesidad de optimizar tanto el motor de simulación de PowerDEVS como la forma de estructurar los modelos. En

particular, en el caso del TDAQ de ATLAS, la estructura altamente interconectada limita la efectividad de estrategias como la paralelización, debido a la fuerte dependencia entre procesos y la complejidad de sincronización.

5.3.1 Análisis de Resultados

Las optimizaciones introducidas en el Capítulo 4 en el motor de simulación y en algunos modelos atómicos de propósito general permitieron alcanzar una mejora significativa en los tiempos de ejecución. Esta mejora se observa en la Figura 5.3 a partir de la métrica RS definida en la Ecuación 3.2. Cada barra en esta figura corresponde al *speedup relativo* entre el tiempo para ejecutar una simulación antes de introducir todos los cambios y el tiempo utilizado luego de aplicadas todas las optimizaciones, con el modelo aplanado y considerando tres tipos de estructuras para el encolamiento de eventos: *Binary Heap* (original),*Pairing Heap*, *Splay Tree*. En términos comparativos, se logró una aceleración relativa de un 66,75 %, lo que evidencia el impacto positivo de las mejoras aplicadas en PowerDEVS.

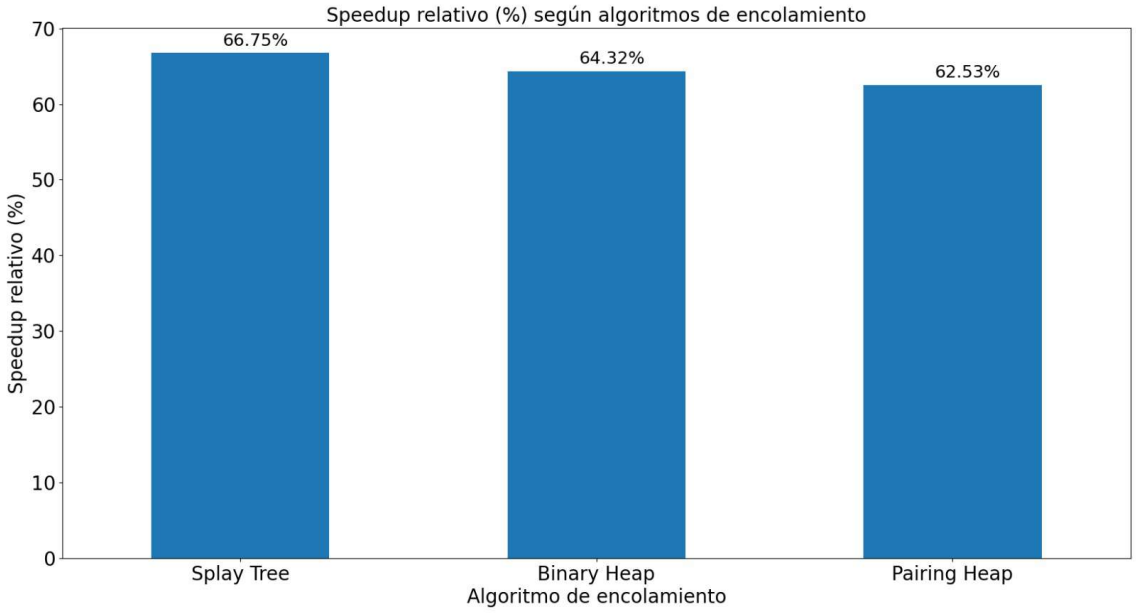


Figura 5.3: Speedups relativos (RS %) para la simulación del modelo TDAQ según algoritmo de encolamiento utilizado.

Antes de aplicar estas optimizaciones, el tiempo promedio necesario para simular tan solo 5 s de tiempo virtual de simulación superaba los 600 s de ejecu-

ción real (tiempo de reloj de pared), lo que representaba una limitación considerable para la realización de experimentos extensivos o iterativos. Con las mejoras incorporadas, este tiempo se redujo a menos de un tercio, situándose en torno a los 198 s en promedio para la misma configuración de prueba. Permitiendo así correr tres veces más experimentos en la misma cantidad de tiempo.

Este avance no solo demuestra la efectividad de las estrategias implementadas —como la optimización de las colas de eventos, la reorganización de estructuras jerárquicas y la revisión de primitivas en modelos atómicos—, sino que además refuerza la viabilidad de emplear PowerDEVS como herramienta práctica en escenarios reales de gran escala y elevada complejidad.

Cabe destacar que para asegurar la consistencia de los resultados, se realizó una verificación detallada —tanto de los resultados de la simulación como de los eventos en el motor del simulador— mediante las nuevas capacidades de *logging* desarrolladas en el marco de este trabajo (Capítulo 3). Gracias a ello, se pudo confirmar que la secuencia de eventos procesada antes y después de aplicar las optimizaciones se mantuvo equivalente, garantizando que la mejora en rendimiento no comprometiera la fidelidad de la simulación.

En síntesis, la reducción sustancial de los tiempos de ejecución obtenida en este caso concreto valida la relevancia de las optimizaciones propuestas y sienta las bases para continuar ampliando las capacidades del simulador hacia escenarios aún más exigentes.

5.4 Modelo MultiSIR para el Análisis de Propagación de Virus

5.4.1 Introducción

En esta sección se evalúa el rendimiento de un modelo epidemiológico de gran escala. Elegimos este modelo porque es apropiado para evaluar la mejora en el rendimiento que puede obtenerse aplicando los cambios presentados en el Capítulo 4 a un modelo híbrido de gran tamaño con fuerte conectividad. Debido a la complejidad de su estructura, este modelo se construyó utilizando

`py2PowerDEVS` [25].

Se trata de una red compuesta por múltiples modelos SIR (*Susceptible-Infectado-Recuperado*), un esquema clásico introducido por Kermack y McKendrick en 1927 [18], que describe la dinámica de propagación de enfermedades infecciosas mediante un conjunto de ecuaciones diferenciales no lineales.

La construcción básica del modelo parte de la definición de un único modelo SIR mediante la interfaz gráfica de PowerDEVS, donde se especifica la evolución de tres compartimentos para las poblaciones: susceptible (S), infectada (I) y recuperada (R). A esta estructura se añade una segunda componente que representa la influencia entre modelos vecinos, permitiendo así simular la propagación del contagio entre distintas regiones o núcleos poblacionales. Las conexiones entre modelos SIR capturan la posibilidad de infecciones cruzadas entre ciudades, barrios o distritos, manteniendo constante la población total en cada nodo.

En este modelo el flujo de infecciones entre nodos está determinado por una matriz de adyacencia Λ, donde cada elemento $\lambda_{i,j}$ representa la tasa de individuos que, estando en el nodo i, se infectan por contacto con individuos del nodo j. Las ecuaciones diferenciales que describen la dinámica de cada nodo consideran tanto la transmisión local como la influencia de nodos vecinos. Para cada región i, se modela la población susceptible S_i, infectada I_i y recuperada R_i, junto con los parámetros de tasa de infección β_i y recuperación γ_i. El comportamiento dinámico del nodo i se muestra en la Ecuación 5.1 y se implementa en PowerDEVS, como muestra el recuadro azul en la Figura 5.4. Además, el recuadro verde representa la influencia de los modelos SIR vecinos.

$$\begin{cases} \dot{S}_i(t) = -\beta_i \cdot S_i(t) \cdot I_i(t) - \sum_{j=1}^{N} \lambda_{i,j} \cdot \beta_i \cdot I_j(t) \cdot S_i(t) \\ \dot{I}_i(t) = \beta_i \cdot S_i(t) \cdot I_i(t) - \gamma_i \cdot I_i(t) + \sum_{j=1}^{N} \lambda_{i,j} \cdot \beta_i \cdot I_j(t) \cdot S_i(t) \quad \forall\, 1 \leq i, j \leq N \\ \dot{R}_i(t) = \gamma_i \cdot I_i(t) \end{cases} \tag{5.1}$$

A partir de este esquema base, se construyó una red de modelos SIR interconectados para representar un conjunto de regiones geográficas independientes,

cada una con su propio comportamiento epidemiológico. Para definir la topología de la red, se utilizaron datos reales de población de radios censales de la Ciudad Autónoma de Buenos Aires y su Área Metropolitana. La disposición geográfica permitió establecer una matriz de adyacencia específica, configurando una estructura de conectividad irregular.

De esta forma, se logró modelar un sistema compuesto por 1,431 nodos SIR interconectados mediante 9,632 enlaces de entrada/salida.

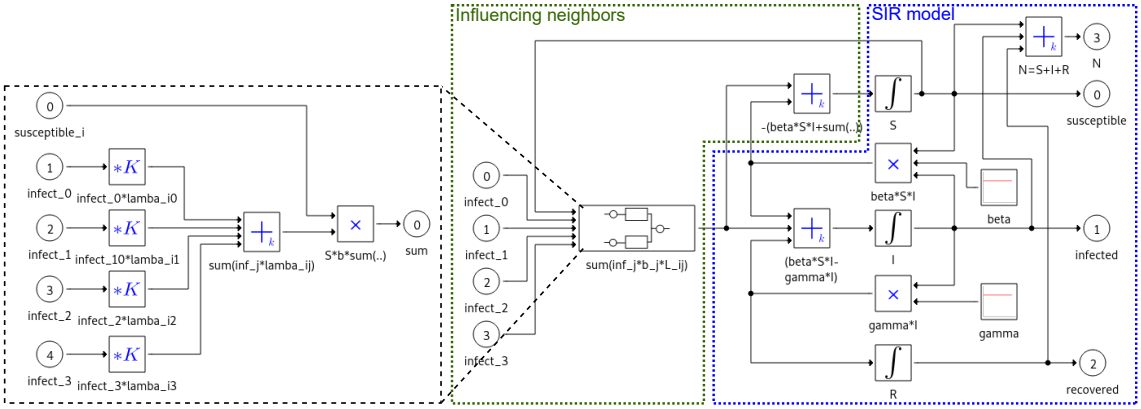


Figura 5.4: Modelo SIR simple para un nodo i-ésimo con $N = 4$ vecinos que lo influencian construido con la GUI PowerDEVS.

5.4.2 Análisis de Resultados

La Figura 5.5 muestra las aceleraciones relativas para el modelo MultiSIR. La mejora en el rendimiento se debe sin duda a la mejora de la gestión de las colas de eventos y las conexiones de entrada/salida, y a los cambios en los integradores de QSS3.

Se puede ver que en este caso, se pudo lograr un speedup relativo de un 9,65 % en el mejor de los casos, utilizando el algoritmo de encolamiento original de PowerDEVS (`Binary Heap`). Sin embargo, la diferencia con utilizar `Splay Tree` es prácticamente despreciable.

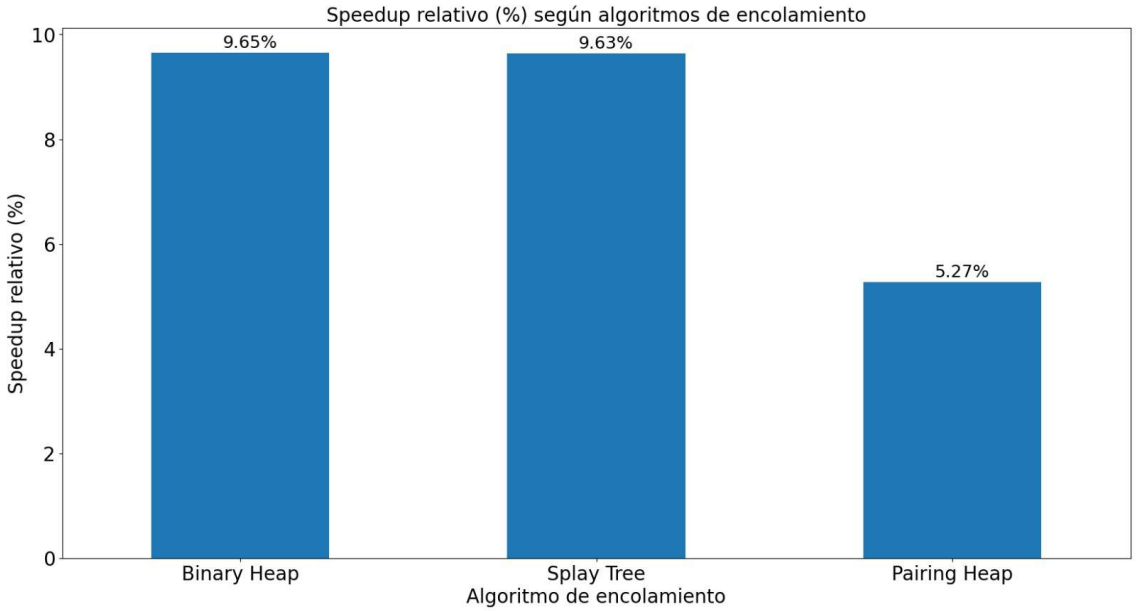


Figura 5.5: Speedups relativos (RS %) promedio para modelo Multi SIR según algoritmo de encolamiento.

5.5 Benchmark Sintético DEVStone

5.5.1 Introducción

Las tres variantes del benchmark sintético DEVStone —LI, HI y HO— descritas en la Sección 2.5.1 se implementaron en PowerDEVS utilizando `py2PowerDEVS` para la generación automatizada de modelos repetitivos, con el fin de evaluar cuantitativamente el impacto de las mejoras de performance aplicadas en el motor de simulación y en la manipulación de estructuras jerárquicas de gran escala. En todos los casos, se decidió utilizar un **retardo de transición externa** de 0, es decir, que los mensajes de entrada de un modelo atómico generan el encolado inmediato de un mensaje de salida del mismo. Asimismo, se decidió utilizar un **retardo de transición interna** de ∞, por lo que los modelos atómicos no agendarían ningún evento sin antes haber recibido un mensaje de entrada. Además, se adoptaron dos configuraciones de eventos de entrada distintas: una con un único evento de entrada en el modelo de mayor jerarquía y otra con un generador que emite eventos periódicos cada 1 s de tiempo de simulación. Esta última configuración permitió evaluar con mayor precisión los tiempos de propagación de los mensajes a través del modelo, minimizando la influencia relativa de la rutina

de inicialización en los tiempos totales de ejecución. En todos los experimentos se emplearon parámetros fijos de **profundidad** $d = 10$ y **ancho** $w = 30$.

5.5.2 Análisis de Resultados

Los resultados de simulación obtenidos antes y después de aplicar las optimizaciones de rendimiento —incluyendo el proceso de achatamiento de modelos y considerando una cola de prioridad *Pairing Heap*— permiten evaluar el impacto concreto de las mejoras sobre cada una de las variantes del benchmark DEVStone. La Figura 5.6 muestra el speedup relativo obtenido. En particular, al comparar los modelos `devstone LI`, `devstone HI` y `devstone HO` bajo una configuración de prueba con un único evento de entrada, se observa que la mayor aceleración relativa fue alcanzada en el modelo HO, con una mejora levemente por encima del 3 %. Este resultado fue seguido de cerca por el modelo HI, que presentó una aceleración promedio cercana al 2 %.

En contraste, el modelo LI mostró una aceleración relativa negativa cuando se considera el tiempo total de ejecución. Este fenómeno se explica por el hecho de que el proceso de achatamiento global se lleva a cabo en tiempo de ejecución, durante la fase de inicialización del modelo. Dado que dicha fase contribuye al tiempo total medido, y considerando que el modelo LI presenta un bajo grado de conectividad interna, el costo adicional de inicialización representa un porcentaje considerable del tiempo total de ejecución, afectando negativamente el resultado global.

Para analizar este efecto con mayor precisión, se consideró una medición separada de los tiempos de ejecución: por un lado, el tiempo total que incluye la inicialización completa del modelo (*tiempo total*), y por otro, una variante donde se excluye específicamente la fase de inicialización del análisis. Al hacer esta distinción, se observó que el modelo LI sí logra una aceleración positiva cuando no se contabiliza el tiempo dedicado al achatamiento (representado por la barra azul en la comparación). Este mismo comportamiento fue identificado también en los modelos HI y HO, cuyos valores de aceleración relativa pasaron de un 2 % y 3 % (barras naranjas) a un 7,2 % y casi 10 % respectivamente (barras azules), al

excluir la inicialización.

Además, en los casos en los que se utilizaron 100 eventos de entrada en el modelo de mayor jerarquía, se puede observar que el beneficio de las optimizaciones es más evidente una vez amortizado el costo inicial asociado a la transformación estructural de los modelos. En particular, las variantes con mayor complejidad topológica —como HI y HO— obtienen un aprovechamiento más efectivo del achatamiento y las mejoras internas al motor, mientras que en modelos simples como LI este tipo de optimizaciones tiene un impacto más limitado si no se realizan múltiples corridas o si no se reúsan instancias ya inicializadas.

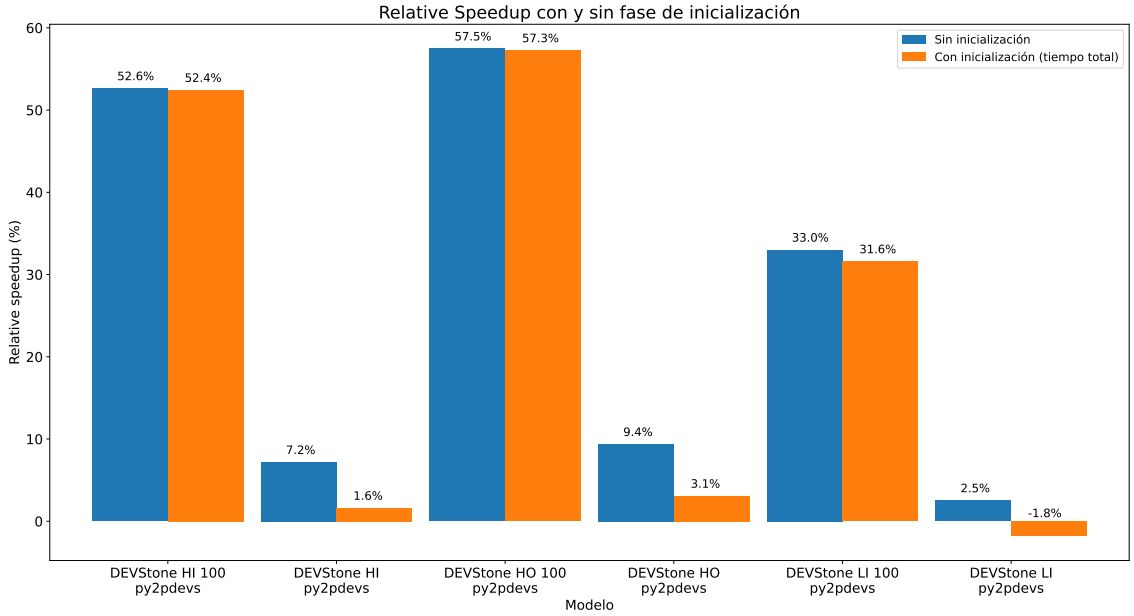


Figura 5.6: Speedups relativos promedio para modelos DEVStone construidos con `py2PowerDEVS`, incluyendo y no incluyendo la fase de inicialización.

Luego, se procedió a analizar el rendimiento de los distintos modelos DEVStone (LI, HI y HO) con 100 eventos de entrada para cada uno de los tres algoritmos de encolamiento. Se puede ver en las Figuras 5.7 y 5.8 que para los modelos LI y HI, el algoritmo de encolamiento que mejor rendimiento reportó fue *Splay Tree*, mientras que *Binary Heap* fue el de menor rendimiento. En la Figura 5.9 el modelo HO, *Pairing Heap* fue levemente superior a *Splay Tree*.

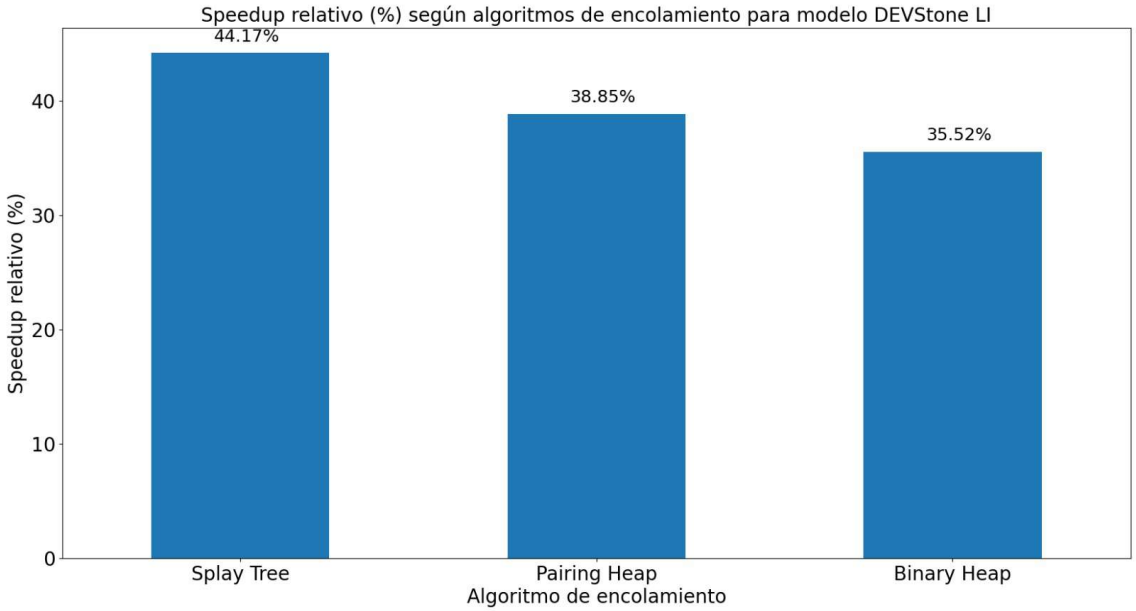


Figura 5.7: Speedups relativos promedio para modelo DEVStone LI según algoritmo de encolamiento.

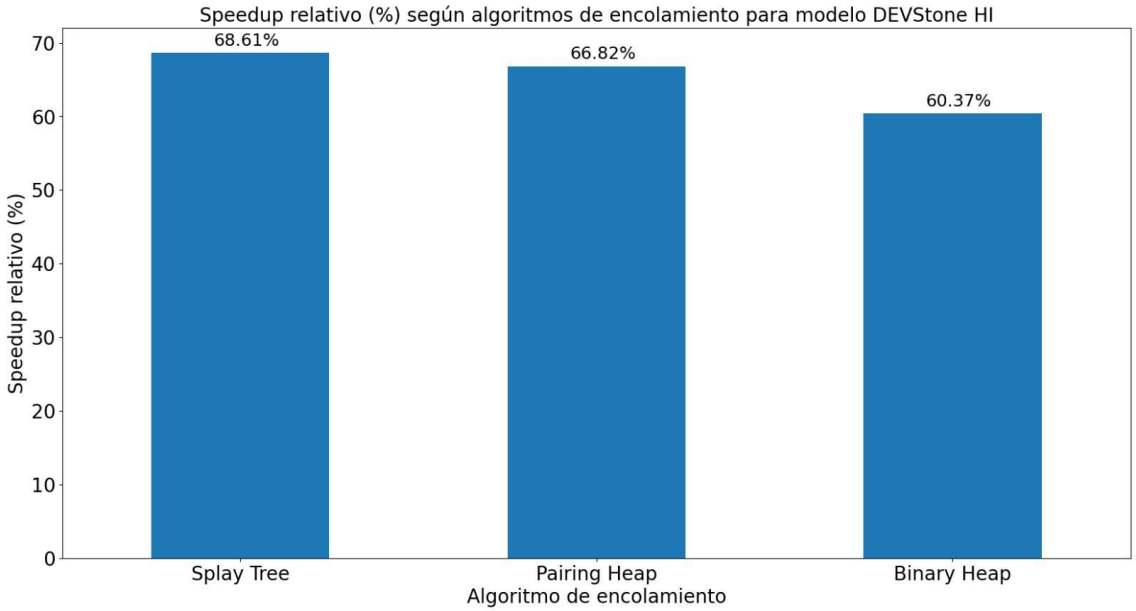


Figura 5.8: Speedups relativos promedio para modelo DEVStone HI según algoritmo de encolamiento.

5.5.3 Generalización Estocástica de DEVStone

Por último, considerando que en numerosos escenarios de simulación la evolución de los modelos presenta componentes de naturaleza aleatoria, se desarrolló una extensión estocástica de las tres variantes clásicas de DEVStone (LI, HI y HO). Esta generalización introduce dos parámetros adicionales, p y λ, que

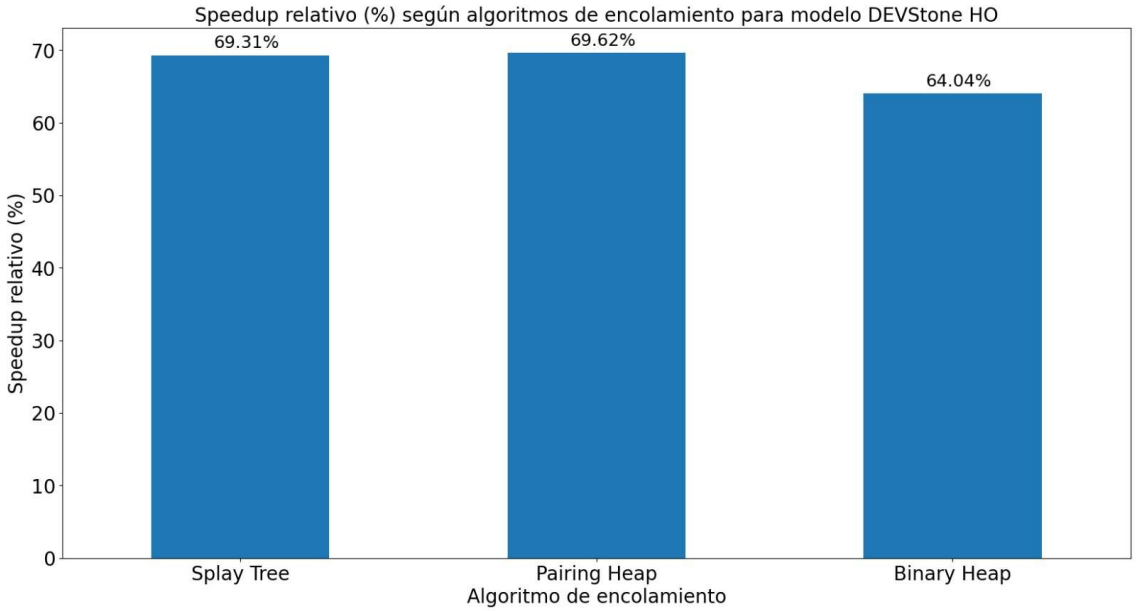


Figura 5.9: Speedups relativos promedio para modelo DEVStone HO según algoritmo de encolamiento.

permiten ajustar el comportamiento probabilístico de los modelos atómicos.

El parámetro p define la proporción de modelos atómicos que generarán una retransmisión con un retardo aleatorio, siguiendo una distribución exponencial de parámetro λ. Por su parte, el complemento de esta proporción, $(1 - p)$, corresponde a la fracción de modelos que retransmitirán un mensaje de forma inmediata, manteniendo la dinámica determinista de las variantes originales.

El propósito de esta generalización es analizar de manera más realista cómo la variabilidad en la distribución temporal de los eventos impacta en la eficiencia de los algoritmos de encolamiento utilizados por el motor de simulación. Esta información resulta valiosa para determinar, en función de las características de cada modelo, cuál es el algoritmo de encolamiento más apropiado.

5.5.3.1 Análisis de Resultados

Se decidió analizar el rendimiento de los distintos algoritmos de encolamiento para la generalización estocástica del modelo DEVStone HO fijando el valor del parámetro $\lambda = 0{,}1$, pero variando el parámetro p. En la Figura 5.10 se puede ver que para $p = 0$, lo que es equivalente al modelo DEVStone HO estándar, *Splay Tree* reporta un rendimiento levemente mejor a *Pairing Heap*, con un *Speedup*

relativo del 67,82 % y 66,35 % respectivamente, mientras que *Binary Heap* reporta 61,1 %. La diferencia entre estos valores y los reportados para el modelo DEVStone HO estándar en la Figura 5.9 puede deberse al uso de un atómico distinto en ambos casos. Para valores de p mayores, se puede ver en las Figuras 5.11 y 5.12 que a medida que incrementa p, los speedups relativos bajan en general para todos los algoritmos de encolamiento, pero lo hacen más exacerbadamente para *Splay Tree* y *Pairing Heap* que para *Binary Heap*, haciendo que este último sea el de mejor rendimiento para $p = 0{,}5$ y $p = 1$. Este resultado indicaría que para modelos más reactivos, es decir, que incluyan mayor proporción de modelos que al recibir un mensaje de entrada, agendan una transición interna en tiempo 0, podría resultar más apropiada la utilización de un *Splay Tree* como cola de prioridad para el scheduling de eventos y para el caso contrario, resultaría más conveniente el uso de un *Binary Heap*. Esto podría deberse a que, al agendar transiciones internas en tiempo 0, y dado que todos los eventos de entrada provienen de un único modelo atómico generador, dichas transiciones se programan en los mismos instantes de tiempo. Este patrón de acceso altamente repetitivo resulta particularmente favorable para el *Splay Tree*. Sin embargo, es necesario un análisis más exhaustivo para poder corroborar dicha hipótesis.

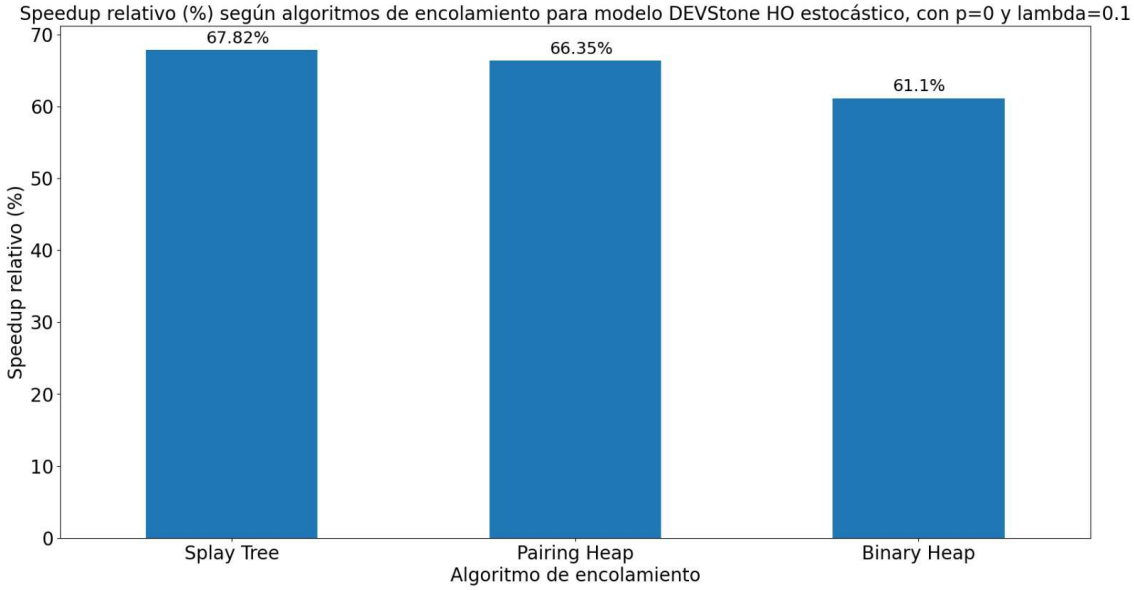


Figura 5.10: Speedups relativos promedio para la generalización estocástica del modelo DEVStone HO con parámetros $p = 0{,}0$ y $\lambda = 0{,}1$ según algoritmo de encolamiento.

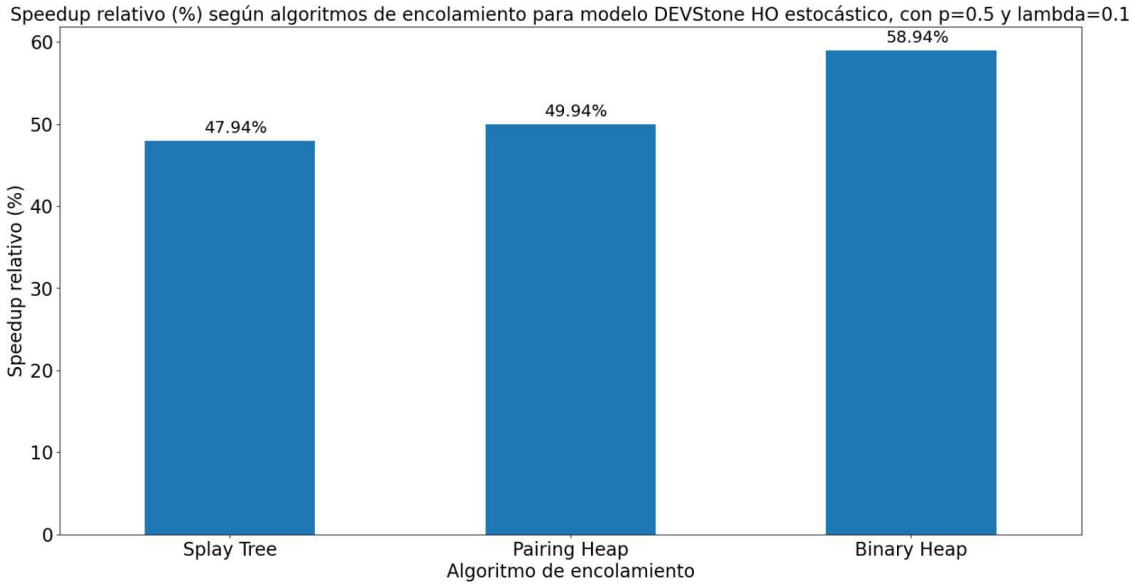


Figura 5.11: Speedups relativos promedio para la generalización estocástica del modelo DEVStone HO con parámetros $p = 0{,}5$ y $\lambda = 0{,}1$ según algoritmo de encolamiento.

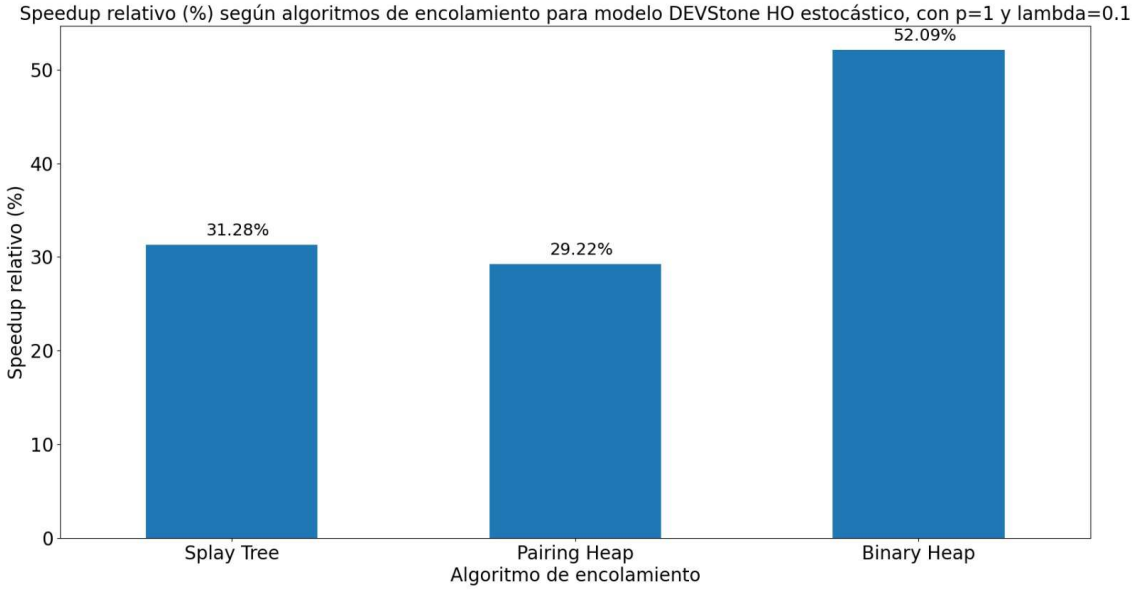


Figura 5.12: Speedups relativos promedio para la generalización estocástica del modelo DEVStone HO con parámetros $p = 1{,}0$ y $\lambda = 0{,}1$ según algoritmo de encolamiento.

6

Conclusiones y Trabajo Futuro

Esta tesis planteó como objetivo central la optimización sistemática del desempeño del simulador PowerDEVS, con un enfoque hacia técnicas que permitan potenciar tanto la eficiencia computacional como la escalabilidad y versatilidad en el modelado y simulación de sistemas complejos a gran escala.

En primer lugar, se abordó la problemática de la eficiencia del motor de simulación. En el estado inicial, PowerDEVS presentaba limitaciones de rendimiento y mantenibilidad al lidiar con modelos que generan grandes volúmenes de eventos o presentan estructuras jerárquicas profundas. Las optimizaciones implementadas permitieron reducir de manera sustancial los tiempos de ejecución, mediante una revisión exhaustiva de los algoritmos internos del motor, la simplificación de trayectorias de datos y la eliminación de redundancias en procesos de scheduling y propagación de eventos. Especial relevancia tuvo la inclusión de colas de prioridad alternativas (Pairing Heap y Splay Tree), que en varios casos de prueba demostraron ser superiores al esquema tradicional, sobre todo bajo cargas intensivas y modelos de alta conectividad. Esta flexibilidad en el scheduling posiciona a PowerDEVS como una herramienta capaz de adaptarse a necesidades específicas, ampliando su aplicabilidad eficiente en modelos de muy variada estructura y dinámica.

En cuanto al análisis y monitoreo del rendimiento, se desarrollaron e inte-

graron herramientas de profiling especialmente adaptadas al ecosistema PowerDEVS. Estas nuevas herramientas reutilizables posibilitaron la identificación de cuellos de botella, favoreciendo la toma de decisiones informadas respecto a intervenciones de optimización. El diseño de comparadores automáticos de consumo de recursos y de resultados entre versiones basados en logs, sumado a la integración de métricas de eficiencia con reporte automático, sentaron las bases para lograr un proceso de mejora continua, incluso en contextos de desarrollo colaborativo y versiones evolutivas del software. Como consecuencia, se hizo posible mostrar –de modo concreto y reproducible– el impacto estadístico de cada mejora propuesta, facilitando tanto la validación interna como la comunicación de resultados.

Desde el punto de vista de la estructura de modelos, el aplanamiento automático representó un salto positivo en la simplificación de la jerarquía interna de los modelos DEVS. El mecanismo desarrollado permite transformar composiciones jerárquicas en representaciones planas, con flexibilidad para elegir el nivel a partir del cual se aplana, optimizando el flujo de eventos y la gestión de conexiones entre bloques. Esta estrategia disminuyó de manera significativa la complejidad implícita en la mensajería interna, lo que se tradujo en menores tiempos de ejecución y una reducción de errores potenciales vinculados a la profundidad de la estructura jerárquica. A nivel de integración y extensibilidad, la adopción de una arquitectura de logging configurable introdujo nuevos mecanismos eficientes de registro que pueden ajustarse dinámicamente a distintas necesidades de depuración, para luego deshabilitarse sin comprometer el rendimiento general.

En el terreno de los modelos atómicos, particularmente relevantes en simulaciones que requieren aproximaciones numéricas precisos, se implementaron varias mejoras. El caso paradigmático fue la revisión del integrador QSS3, donde las herramientas de profiling permitieron identificar un cuello de botella en la función utillizada para el cálculo de raíces cúbicas, cuyo reemplazo mejoró la eficiencia y exactitud de la integración por cuantificación de estados. Asimismo, la utilización de nuevas estructuras de datos en modelos DataLoggers optimizó el cálculo de métricas estadísticas, elemento fundamental para el análisis y

validación de simulaciones extensas. En los modelos de redes de datos, la reducción de la sobrecarga generada por el registro de ciertas variables incidió directamente en el rendimiento de escenarios a gran escala, donde el acceso y procesamiento masivo de información era antes un factor limitante.

Una validación esencial de las propuestas presentadas se realizó a través de estudios de caso de relevancia y escala, tales como la simulación del sistema TDAQ del experimento ATLAS en el CERN. En estos escenarios, se pudo observar aceleraciones relativas superiores al 60 % en comparación con versiones previas del software. La batería de pruebas incluyó tanto modelos reales como sintéticos y abarcó distintas topologías y requisitos, demostrando la generalidad de los aportes.

De este modo, a través de la combinación de optimizaciones en el motor, mejoras en el soporte a desarrolladores y usuarios mediante herramientas de análisis, y la reformulación de componentes clave a nivel de modelos atómicos, la presente Tesis logró alcanzar los objetivos propuestos. PowerDEVS no sólo incrementó su eficiencia, sino también su flexibilidad y su horizonte de aplicabilidad.

Finalmente, cabe remarcar que los métodos sistematizados aquí desarrollados resaltan la importancia de combinar estrategias de bajo nivel (algorítmicas y estructurales) con soluciones de apoyo para el monitoreo y la validación. Esto permite pensar en un ciclo virtuoso de optimización continua, donde el propio proceso de medición y comparación incentiva nuevas iteraciones de mejora. Se sientan así bases firmes para futuras evoluciones del simulador y su aplicación en problemáticas cada vez más complejas y demandantes, abriendo la puerta a líneas de trabajo que incluyan, por ejemplo, la exploración de técnicas de paralelización, generación automática de modelos y adaptaciones a nuevas arquitecturas de hardware emergentes.

En síntesis, la tesis no solo contribuye con soluciones concretas y mensurables, sino que también provee un marco metodológico y experimental para la optimización sistemática de simuladores de eventos discretos de uso general.

En cuanto al trabajo futuro, se continuará utilizando las herramientas desarrolladas en este trabajo en la búsqueda de optimizaciones adicionales del

rendimiento de PowerDEVS. Asimismo, se integrará en el repositorio de PowerDEVS la batería de modelos seleccionados para validar de forma automática los commits en un proceso de integración continua. Por otro lado, se utilizará la implementación del benchmark DEVStone construido en este trabajo para comparar el rendimiento de PowerDEVS con el de otros simuladores DEVS populares. Asimismo, se incorporará al sistema de logging del motor de simulación la posibilidad de seleccionar tanto el formato de salida —ya sea orientado a la interpretación humana o a la comparación automática— como el tipo de archivo generado, con el objetivo de mejorar su usabilidad y su eficiencia. Por último, se observó que el algoritmo de manejo de colas de prioridad que permite lograr un mejor rendimiento depende del modelo. En el último capítulo presentamos un análisis exploratorio del uso de distintos algoritmos de encolamiento para distintas parametrizaciones del modelo DEVStone estocástico. No obstante, estos resultados no son concluyentes y ameritan un análisis más exhaustivo para poder determinar de antemano el algoritmo más adecuado para un dado modelo, así como una futura exploración de otros algoritmos alternativos para el manejo de colas de eventos en el motor del simulador de PowerDEVS. Más aún, un aspecto para analizar a futuro es la posibilidad de utilizar distintos algoritmos de encolamiento por modelo acoplado en un modelo no aplanado.

Bibliografía

[1] Andrade, L. Conversión de modelos powerdevs al lenguaje modelica. B.S. thesis, Facultad de Ciencias Exactas, Ingeniería y Agrimensura. Universidad Nacional . . . , 2015.

[2] Bergero, F., and Kofman, E. PowerDEVS: A tool for hybrid system modeling and real-time simulation. *Simulation 87*, 1-2 (2011), 113–132.

[3] Bergero, F., and Kofman, E. A vectorial devs extension for large scale system modeling and parallel simulation. *SIMULATION 90*, 5 (2014), 522–546.

[4] Bonaventura, M. *Modelado y Simulación Híbrida de Redes Complejas de Datos*. PhD thesis, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2019.

[5] Bonaventura, M., Foguelman, D., and Castro, R. Discrete event modeling and simulation-driven engineering for the atlas data acquisition network. *Computing in Science & Engineering 18*, 3 (2016), 70–83.

[6] BoostLib. Boost.python library. Available at `https://www.boost.org/`, accessed 18th April, 2022.

[7] Cárdenas, R., Henares, K., Arroba, P., Risco-Martín, J. L., and Wainer, G. A. The devstone metric: Performance analysis of devs simulation engines. *ACM Transactions on Modeling and Computer Simulation 32*, 3 (2022), 1–20.

[8] Castro, R., Bergonzi, M., Marcosig, E. P., Fernández, J., and Kofman, E. Discrete-event simulation of continuous-time systems: evolution and state

of the art of quantized state system methods. *SIMULATION 100*, 6 (2024), 613–638.

[9] Castro, R., Kofman, E., and Cellier, F. E. Quantization-based integration methods for delay-differential equations. *Simulation Modelling Practice and Theory 19*, 1 (2011), 314–336. Publisher: Elsevier.

[10] Castro, R., Kofman, E., and Cellier, F. E. Quantization-based integration methods for delay-differential equations. *Simulation Modelling Practice and Theory 19*, 1 (2011), 314–336. Modeling and Performance Analysis of Networking and Collaborative Systems.

[11] Castro, R., Kofman, E., and Wainer, G. A Formal Framework for Stochastic Discrete Event System Specification Modeling and Simulation. *SIMULATION 86*, 10 (2010).

[12] Cellier, F. E., and Kofman, E. *Continuous System Simulation*, 1 ed. Springer Science & Business Media, New York, NY, USA, 2006.

[13] Floros, X., Bergero, F., Cellier, F. E., and Kofman, E. Automated simulation of modelica models with QSS methods-the discontinuous case. In *Proceedings of the 8th International Modelica Conference* (2011), pp. 657–667.

[14] Fredman, M. L. On the efficiency of pairing heaps and related data structures. *J. ACM 46*, 4 (July 1999), 473–501.

[15] Fredman, M. L., Sedgewick, R., Sleator, D. D., and Tarjan, R. E. The pairing heap: A new form of self-adjusting heap. *Algorithmica 1*, 1–4 (Jan. 1986), 111–129.

[16] Glinsky, E., and Wainer, G. Devstone: a benchmarking technique for studying performance of devs modeling and simulation environments. In *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT)* (2005), Oct. 10th-12th, NW Washington, DC, USA, 265-272.

[17] Jones, D. W. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM 29*, 4 (1986), 300–311.

[18] Kermack, W. O., McKendrick, A. G., and Walker, G. T. A contribution to the mathematical theory of epidemics. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character 115*, 772 (1927), 700–721.

[19] Kofman, E. A second-order approximation for DEVS simulation of continuous systems. *Simulation 78*, 2 (2002), 76–89.

[20] Kofman, E. A third order discrete event method for continuous system simulation. *Latin American applied research 36*, 2 (2006), 101–108.

[21] Kofman, E. Relative error control in quantization based integration. *Latin American applied research 39*, 3 (2009), 231–237.

[22] Kofman, E., and Junco, S. Quantized-state systems: a DEVS Approach for continuous system simulation. *Transactions of The Society for Modeling and Simulation International 18*, 3 (2001), 123–132.

[23] Lanuza, J., Trabes, G. G., and Wainer, G. A. Parallel execution of devs in shared-memory multicore architectures. In *Proceedings of 2020 Spring Simulation Conference (SpringSim)* (2020), May 19th-21th, Fairfax, VA, USA, 1-11.

[24] Pecker-Marcosig, E., Romczyk, G., Bonaventura, M., and Castro, R. Systematic performance optimization for the powerdevs simulator and models of large-scale real-world applications. In *2024 Winter Simulation Conference (WSC)* (2024), pp. 2797–2808.

[25] Pecker–Marcosig, E., Bonaventura, M., Lanzarotti, E., Santi, L., and Castro, R. py2powerdevs: Construction and manipulation of large complex structures for powerdevs models via python scripting. In *2022 Winter Simulation Conference (WSC)* (2022), pp. 2594–2605.

[26] Pozo Astigarraga, E., Bonaventura, M., Maple, J., Pecker Marcosig, E., Levrini, G., and Castro, R. D. Benchmarking data acquisition event building network performance for the atlas hl-lhc upgrade. In *Proc. of the 26th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2023)* (2023), May 8th-12th, Norfolk, VA, USA, 1-11.

[27] Risco-Martín, J. L., Mittal, S., Jiménez, J. C. F., Zapater, M., and Correa, R. H. Reconsidering the performance of devs modeling and simulation environments using the devstone benchmark. *Simulation 93*, 6 (2017), 459–476.

[28] Santi, L., Rossi, L., and Castro, R. Efficient discrete-event based particle tracking simulation for high energy physics. *Computer Physics Communications 258* (2021), 107619.

[29] SEDLab. Repositorio GIT de PowerDEVS. Disponible en `https://git-modsimu.exp.dc.uba.ar/matiasb/powerdevs-CERN/`, accedido 8th Julio, 2025.

[30] Sleator, D. D., and Tarjan, R. E. Self-adjusting binary search trees. *J. ACM 32*, 3 (July 1985), 652–686.

[31] Van Tendeloo, Y., and Vangheluwe, H. An Evaluation of DEVS Simulation Tools. *Simulation 93*, 2 (2017), 103–121.

[32] Vangheluwe, H. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *Proceedings of the IEEE International Symposium on Computer-Aided Control System Design* (Sept 2000), CACSD'00, IEEE, pp. 129–134.

[33] Wainer, G. A. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*, 1st ed. CRC Press, Inc., USA, 2009.

[34] Weicker, R. P. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM 27*, 10 (Oct. 1984), 1013–1030.

[35] Zeigler, B. P., and Lee, J. S. Theory of Quantized Systems: Formal Basis for DEVS/HLA Distributed Simulation Environment. In *Enabling Technology for Simulation Science II* (1998), A. F. Sisti, Ed., vol. 3369, International Society for Optics and Photonics, SPIE, pp. 49 – 58.

[36] Zeigler, B. P., Muzy, A., and Kofman, E. *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*, 3 ed. Academic Press, San Diego, CA, USA, 2018.