Evaluación del Rendimiento de Simuladores de Eventos Discretos

Beatriz Herguedas Pinedo

Universidad Complutense de Madrid

Facultad de Informática

Departamento de Arquitectura de Computadores y Automática

Madrid, 21 de septiembre de 2021

Dirigido por: JOSÉ LUIS RISCO MARTÍN
Colaborador: KEVIN HENARES VILABOA

A mi familia, especialmente a mi hermana Cristina, por estar siempre.

A mis queridos Arturo y Jorge, por impulsarme y creer en mí.

Resumen

El presente trabajo estudia y propone una métrica novedosa para la evaluación del rendimiento de simuladores de eventos discretos. En particular, nos centraremos en la evaluación de simuladores que implementan el formalismo DEVS, y nos basaremos en el enfoque hasta ahora propuesto a través del *benchmark* DEVStone.

Abstract

In this work, we study and propose a novel metric for the evaluation of performance of discrete event simulators. In particular, we focus on analysing simulators which implement the DEVS formalism, and we will use the approach already proposed in the DEVStone benchmark.

Índice general

Capítulo 1

Introducción

En este trabajo cubriremos el estudio del rendimiento de diferentes simuladores que siguen el formalismo Discrete Event System Specification (DEVS). Hasta ahora existían propuestas de *benchmarks*, como DEVStone, que ofrecían posibles tests a ejecutar en simuladores de eventos discretos. Sin embargo, este trabajo busca ampliar la metodología existente para proponer también el uso de métricas para la evaluación de los resultados obtenidos. De esta manera, se propondrá una forma novedosa para el análisis y la comparativa de los sistemas basados en DEVS.

1.1. Motivación

En el mundo de la simulación la mayoría de sistemas se modelan mediante eventos discretos [12]. Entre los diferentes formalismos propuestos para el modelaje de estos sistemas se encuentra DEVS [50], un enfoque ampliamente utilizado en el propio departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid.

En la actualidad existen multitud de implementaciones de DEVS a través de librerías y simuladores. Sin embargo, y a pesar del uso extendido de este tipo de sistemas, no existe un marco común donde se analicen los simuladores disponibles y su rendimiento. Debido a esto, en los últimos años se han propuesto diferentes aproximaciones para poder realizar este análisis [35, 21]. En general, esta tarea propuesta no es sencilla, y existen

multitud de investigaciones referidas al estudio de rendimiento en sistemas computacionales que ponen de relieve la problemática existente [19, 37].

En este trabajo se buscará asentar una estrategia fija para la evaluación del rendimiento de los simuladores basados en DEVS, proponiendo métricas con las cuales poder determinar qué simulador ofrece, en términos de rendimiento, un mejor comportamiento. Así, se propone evaluar la forma en la que estos sistemas están construidos para proponer una métrica adecuada para el análisis de los datos. A su vez, y para complementar el examen teórico de la idoneidad del estimador elegido, también se busca poder valorar los resultados arrojados por esta métrica empíricamente.

1.2. Objetivos

Podemos categorizar los objetivos de este trabajo como generales o específicos. Los objetivos generales están orientados al desarrollo de metas a gran escala. Entre estas podemos destacar la búsqueda de examinar y aprender sobre los simuladores más importantes que hay en la literatura, siendo capaz de analizarlos y obtener resultados numéricos que cuantifiquen su rendimiento.

Por el contrario, los objetivos específicos profundizan más en las diferentes cuestiones que se abordan. Por un lado, buscaremos que los simuladores analizados sean representativos de entre la gran variedad existente, así como de las implementaciones en los diferentes lenguajes de programación disponibles. A su vez, también se marca como objetivo específico el examen profundo de técnicas de análisis de datos para comprender su viabilidad y aplicabilidad.

Podemos listar pormenorizadamente los objetivos generales y específicos tal y como sigue:

[O1] Estudiar el formalismo DEVS y examinar las diferentes librerías o simuladores disponibles en la literatura.

- [O1.1] Definir la noción de sistema y simulador de eventos discretos.
- [O1.2] Definir y describir los sistemas basados en DEVS.

[O1.3] Indagar sobre las diferentes librerías y simuladores disponibles, exponiendo una muestra representativa de implementaciones en diferentes lenguajes de programación.

[O2] Investigar sobre el análisis de rendimiento de sistemas y las diferentes técnicas empleadas en el campo.

[O2.1] Definir el concepto de *benchmark* y estudiar sus principales características.

[O2.2] Exponer el concepto de métrica de rendimiento y analizar las diferentes métricas más utilizadas, profundizando en la aplicabilidad y limitaciones.

[O3] Describir el *benchmark* DEVStone para sistemas DEVS, detallando los diferentes modelos y su aplicabilidad.

[O3.1] Detallar el funcionamiento de DEVStone y su utilidad.

[O3.2] Exponer y describir los principales modelos disponibles en los generadores de modelos sintéticos en DEVStone.

[O4] Proponer una métrica de rendimiento para la evaluación de simuladores de eventos discretos basados en DEVS.

[O4.1] Estudiar las diferentes técnicas y métricas utilizadas para cuantificar el rendimiento de sistemas, detallando cuál es la más adecuada para la evaluación de simuladores en DEVS.

[O4.2] Evaluar el alcance y aplicabilidad de la métrica, proponiendo un caso real de aplicación donde pueda estudiarse su utilidad.

[O4.3] Exponer técnicas complementarias, cualitativas o cuantitativas, que puedan ayudar a entender, o apoyen, las conclusiones que se puedan derivar de la métrica de rendimiento.

[O5] Comparar el rendimiento de algunos de los principales simuladores e implementaciones de DEVS.

[O5.1] Proponer un marco de trabajo adecuado para la ejecución de instancias que permitiese comparar correctamente el desempeño de distintos simuladores.

[O5.2] Analizar los simuladores más relevantes utilizando diferentes implementaciones, modelos y métricas de rendimiento.

[O5.3] Concluir cuáles son las principales diferencias entre los distintos simuladores: cuáles son más rápidos en la ejecución de instancias simples o complejas y cuáles realizan una mejor gestión de los recursos de memoria.

[O6] Estudiar el alcance y las limitaciones del trabajo, así como proponer futuras vías de investigación para continuarlo.

1.3. Estructura del documento

El documento se encuentra dividido en cinco capítulos. El capítulo 1 introduce el concepto de simuladores de eventos discretos, el formalismo DEVS y la motivación del trabajo. A su vez, se incluye una lista de objetivos generales y específicos que se buscan completar.

Seguidamente, en el capítulo 2 se indaga en el campo de los sistemas y los simuladores de eventos discretos, detallando el funcionamiento de los sistemas basados en DEVS y cubriendo sus diferentes implementaciones en los principales lenguajes de programación más utilizados hoy en día. Tras esto, se procede a abordar el estudio del *benchmarking*, exponiendo las principales técnicas empleadas en la medida del rendimiento de sistemas. Además, se detallan las propiedades matemáticas de las diferentes métricas empleadas. Finalmente, para concluir este capítulo, se incluye una descripción del *benchmark* DEVStone, examinando los principales modelos utilizados y sus complejidades de cómputo.

Por otro lado, en el capítulo 3, se delimita la metodología de trabajo para evaluar las diferentes implementaciones de DEVS en simuladores y librerías. Así, se expone primero el marco de trabajo en el cual se operará, pormenorizando los detalles referidos al entorno de ejecución y los simuladores empleados en las pruebas. Tras esto, se exponen las técnicas que se emplearán para el análisis del rendimiento, estudiando su aplicabilidad y proponiendo una estrategia que permita evaluar los resultados.

En el capítulo 4 se recogen los resultados del análisis utilizando la metodología descrita en el capítulo anterior. Además, se incluye una descripción

de las ejecuciones, detallando los errores y problemas encontrados y la solución adoptada de cara al análisis de los datos. También, se añaden representaciones gráficas de los resultados a través de mapas de calor, ofreciendo un enfoque diferente para examinar y comparar los simuladores. Para concluir el capítulo, se incluyen los resultados numéricos de las métricas y estimadores calculados para la población de datos dada por las simulaciones. Estos resultados son además evaluados, permitiendo una comparación entre los simuladores.

Por último, en el capítulo 5 se revisan los objetivos marcados al inicio del trabajo, y se resumen las principales conclusiones extraídas de este estudio. A su vez, se proponen también diferentes vías de estudio que permitan continuar con la investigación del rendimiento de sistemas basados en DEVS.

Capítulo 2

Estado del arte

En este capítulo cubriremos las nociones básicas para abordar de forma efectiva los simuladores de eventos discretos y el formalismo DEVS. Para ello, comenzaremos introduciendo de manera general el concepto de sistema y modelos de eventos discretos, para posteriormente exponer la propuesta de modelización de sistemas utilizando DEVS.

Tras esto, pasaremos a exponer las principales implementaciones de DEVS hasta ahora existentes, que nos serán de particular interés en futuros capítulos donde emplearemos estas implementaciones para ofrecer una propuesta de *benchmarking* para simuladores de eventos discretos. De esta manera, continuaremos el capítulo estudiando distintas técnicas de evaluación de rendimiento y *benchmarking* que puedan resultar atractivas, comparando y estudiando métricas como las medias pitagóricas y su interés en el análisis de datos. Finalmente, concluiremos introduciendo los modelos DEVSstone, que aspiran a poder ofrecer una base sobre la que construir evaluaciones de rendimiento de los modelos en DEVS y que, de hecho, sustentarán la propuesta expuesta en posteriores capítulos.

2.1. Simuladores de eventos discretos y DEVS

En esta sección cubriremos las nociones de sistemas, simuladores de eventos discretos y el formalismo DEVS. A su vez, cubriremos algunas de las principales implementaciones de este formalismo que serán posteriormente

empleadas durante el desarrollo de este trabajo.

2.1.1. Sistemas y modelos de eventos discretos

Comenzaremos esta subsección introduciendo el concepto de *sistema*. Formalmente, coincidiremos en la definición de sistema propuesta por Wymore [47], que intuitivamente lo describe como un conjunto compuesto por una entrada, un proceso que puede alterar la misma y que depende de un cierto estado y, en general, una salida. La figura 2.1 muestra cómo puede representarse gráficamente un sistema.

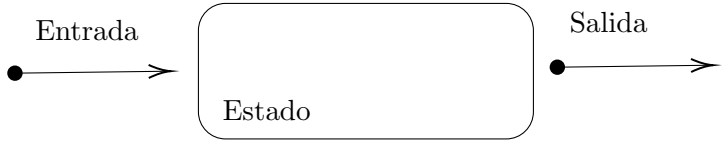


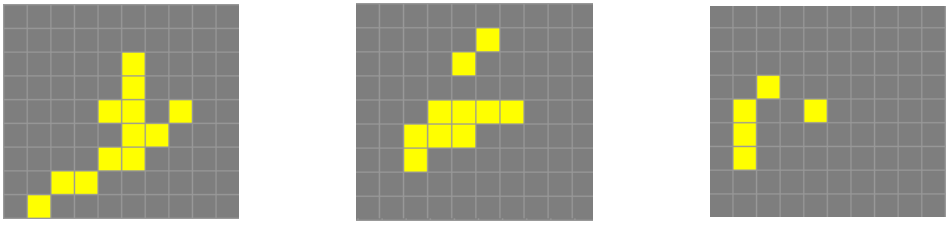
Figura 2.1: Representación de un sistema

El concepto de sistema es en general amplio, y puede ser utilizado para capturar multitud de escenarios. A modo de ejemplo, pueden verse como sistemas todas aquellas instancias que pueden representarse por las conocidas máquinas de Moore y Mealy. Además, debido a la formalización subyacente en los sistemas de Wymore, es posible proceder al análisis de la verificación y corrección de implementaciones de modelos que emulen estos sistemas. Siguiendo precisamente esta estrategia, se publica en 1976 la primera edición de *Theory of Modeling and Simulation* [50]. Este libro expone los fundamentos de la Teoría del Modelizado y Simulación de Sistemas, permitiendo construir modelos formales de sistemas complejos.

En particular, con el surgimiento de formalismos que recogen la simulación y el modelaje de sistemas, se empiezan a estudiar aquellos sistemas dinámicos cuya variación depende de eventos discretos, esto es, sistemas de *Simulación de Eventos Discretos* (la abreviatura se hereda del inglés, DES, Discrete Event Simulation). Los DES se basan en el análisis de una secuencia de eventos que acontecen en el tiempo, teniendo lugar cada evento en un momento determinado y definiendo un cambio de estado en el sistema.

Un ejemplo clásico de DES es el Juego de la vida diseñado por el matemático John Horton Conway [20]. Este sistema puede definirse como un

autómata celular, un modelo dinámico que evoluciona en pasos discretos. En el Juego de la vida, el estado está definido por una colección de celdas que pueden o no albergar una célula viva. A su vez, se definen una serie de reglas que establecen el comportamiento de cada celda en función del estado de las celdas adyacentes. La figura 2.2 ejemplifica este sistema en diferentes estados.



Estado inicial Estado intermedio Estado final

Figura 2.2: Ejemplo de tres iteraciones en el Juego de la vida, las celdas amarillas destacan células vivas y las grises celdas vacías.

2.1.2. El formalismo DEVS y los simuladores de eventos discretos

El formalismo de DEVS (del inglés, Discrete Event System Specification, esto es, Especificación de Sistemas de Eventos Discretos), puede definirse como una subclase de los sistemas de Wymore que captura las características relevantes para los modelos de simulación orientados a eventos [49]. De esta manera, este formalismo especifica qué es un modelo y qué debe o no contener el mismo. Este paradigma es además universal y único para los modelos de sistemas de eventos discretos, esto es, cualquier sistema que acepte eventos como entradas en el tiempo es equivalente a uno en DEVS (o lo que es lo mismo, su estructura y comportamiento pueden describirse también usando este formalismo).

Existen dos variantes del formalismo DEVS: la variante clásica (Classic DEVS) y la variante paralela (Parallel DEVS), que mantiene las características útiles del DEVS Clásico y elimina las restricciones de serialización que impiden llevar a cabo una ejecución múltiple en un entorno paralelo

[17]. En este trabajo se hace siempre referencia a la variante paralela, y de ahora en adelante cuando hablemos de DEVS nos referiremos a dicha variante.

Generalmente, en DEVS se representan los sistemas a través de tres conjuntos diferentes y cinco funciones. Introducimos a continuación la notación básica que utilizaremos en este trabajo:

- El conjunto de entrada, X.
- El conjunto de salida, Y.
- El conjunto de estados, S.
- La función de transición externa, δ_{ext}.
- La función de transición interna, δ_{int}.
- la función de confluencia, δ_{con}.
- La función de salida, λ.
- La función de avance temporal, ta.

Por otro lado, desde la óptica del formalismo de DEVS, los sistemas se pueden dividir en modelos básicos, denominados atómicos, o modelos compuestos, también llamados acoplados. Los modelos atómicos definen el comportamiento de un sistema, mientras que los acoplados especifican su estructura [35]. Formalmente, podemos expresar los modelos atómicos como:

Definición 2.1.1 (Modelo atómico)**.** Se define un modelo atómico como la tupla

$$A = \langle X;\ Y;\ S;\ \delta_{ext};\ \delta_{int};\ \delta_{con};\ \lambda;\ ta \rangle$$

donde X, Y, S, δ_{ext}, δ_{int}, δ_{con}, λ y ta se corresponden con los símbolos de conjuntos y funciones de DEVS especificados anteriormente.

Los modelos atómicos en DEVS procesan la entrada de eventos basándose en el estado actual y su condición, generando eventos de salida y desembocando en transiciones a otros estados de acuerdo al estado actual. A

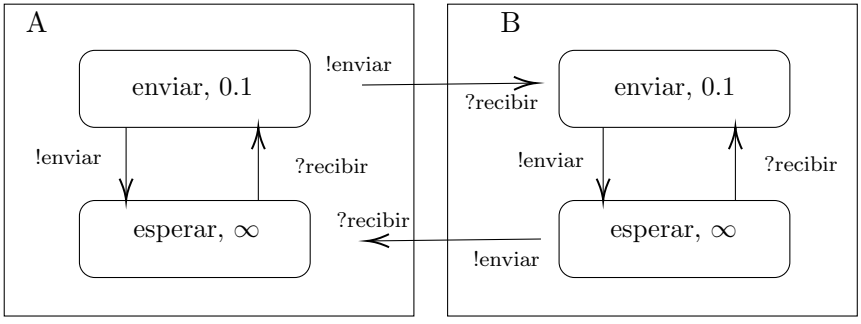


Figura 2.3: Representación del juego de Ping Pong en DEVS

modo de ejemplo, la figura 2.3 muestra cómo puede construirse un modelo atómico en DEVS que capture el comportamiento del juego del Ping Pong.

Notemos entonces que el juego del Ping Pong, expresado en forma de modelo atómico en DEVS, se describe a partir de dos jugadores A y B. Ambos jugadores tienen dos estados posibles, *enviar* o *esperar*. Para enviar se necesitan 0.1 segundos para poder devolver la bola a la salida a través del evento *!enviar*, mientras que el estado *esperar* tiene una duración de tiempo indefinida, hasta que el jugador recibe la pelota a través del evento *?recibir*. De esta manera, la estructura del juego es sencilla. El jugador A emite un evento de salida que se transmite al jugador B y viceversa.

Para este juego es posible entonces proporcionar explícitamente los siguientes valores para los conjuntos y funciones que definen el modelo atómico:

- $X = \{?recibir\}$
- $Y = \{!enviar\}$
- $S = \{(d, \sigma) : d \in \{esperar, enviar\},\ \sigma \in \mathbb{T}^{\infty}\}$, donde $\mathbb{T}^{\infty}\} = [0, \infty]$
- $s_0 = (enviar, 0.1)$
- $ta(s) = \sigma,\ \forall s \in S$
- $\delta_{ext}(((esperar, \sigma), t_e), ?recibir) = (enviar, 0.1)$
- $\delta_{int}(enviar, \sigma) = (esperar, \infty)$

- $\delta_{int}(esperar, \sigma) = (enviar, 0.1)$
- $\lambda(enviar, \sigma) = !enviar$
- $\lambda(esperar, \sigma) = \emptyset$

De forma análoga al caso anterior, podemos introducir formalmente el concepto de modelo acoplado:

Definición 2.1.2 (Modelo acoplado)**.** Se define un modelo acoplado como la tupla

$$M = \langle X;\ Y;\ C;\ EIC;\ EOC;\ IC \rangle$$

donde X es el conjunto de entrada, Y el de salida, C es un conjunto de modelos atómicos/acoplados, EIC es el conjunto que define la relación de acoplamiento externa (qué entradas externas de M se relacionan con componentes de C), EOC es la relación de acoplamiento externa (esto es, define la relación entre las salidas de los componentes de C con las salidas de M) y, finalmente, IC es la relación de acoplamiento interna (expresa las relaciones de salida de componentes $c_i \in C$ con las salidas de $c_j \in C$ para $i \neq j$).

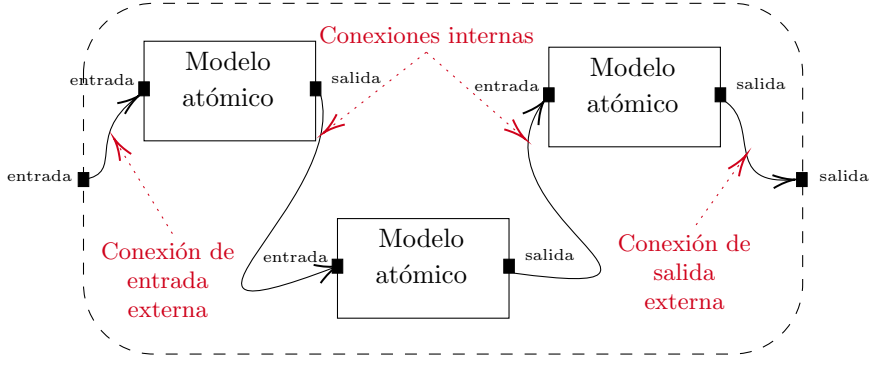


Figura 2.4: Representación de un modelo acoplado.

A modo de ejemplo, la figura 2.4 representa un modelo acoplado sencillo y sus interconexiones detalladas.

Notemos, además, que los modelos acoplados pueden también componerse y agregarse mutuamente, al igual que se combinan con modelos atómicos.

Por otro lado, utilizando el formalismo anterior es posible construir simuladores que ejecuten modelos en DEVS de forma correcta y eficiente. Para ello, podemos aprovecharnos del cierre bajo el acoplamiento que garantiza la corrección en la composición jerárquica de los modelos. Precisamente es esta propiedad la que resulta más atractiva para la implementación DEVS de simuladores, pudiendo además desplegarse en cualquier plataforma. Así, los simuladores de este tipo han sido empleados con éxito en multitud de dominios de diferente índole. A modo de ejemplo, podemos destacar su uso en el rendimiento y consumo energético del despliegue de micro centros de datos [15], en el entorno para el análisis predictivo de eventos críticos en pacientes de Ictus [23], o en el modelado de la producción de energía eólica [34].

2.1.3. Implementaciones de DEVS

En los últimos años han surgido multitud de motores que implementan DEVS, que ofrecen interfaces de programación de aplicaciones (del inglés, Application Programming Interfaces o APIs) y permiten definir modelos usando lenguajes de alto nivel. Algunas de las implementaciones más utilizadas se recogen a continuación.

A Discrete Event System simulator, aDEVS

aDEVS es una librería desarrollada por Jim Nutaro en C++, y cuyo objetivo es permitir el desarrollo de simulaciones de eventos discretos basadas en implementaciones de los formalismos DEVS y Dynamic DEVs [1].

La librería está disponible para su descarga en el repositorio abierto SourceForge[1], pudiéndose encontrar todas las diferentes versiones estables publicadas hasta la fecha [2].

[1] `https://sourceforge.net/`

PyPDEVS

PythonPDEVS, también abreviado como PyPDEVS, implementa DEVS en el lenguaje de programación Python. Las versiones de los últimos años de PyPDEVS están sobre todo centradas en mejorar el rendimiento, principalmente por el hecho de que Python es un lenguaje interpretado [39, 40].

La librería puede encontrarse en el sitio web MSDL Git[2], que ofrece repositorios donde alojar y gestionar proyectos. Además, en la página donde está disponible se detalla su proceso de instalación y configuración [48].

xDEVS

xDEVS es una librería que contiene un conjunto de clases en C++, Java y Python que ofrecen una interfaz para la ejecución de simulaciones orientadas a eventos. Esta interfaz implementa el formalismo de DEVS, siendo el principal objetivo del proyecto poder simular modelos en tiempo virtual y real, y ejecutar simulaciones secuenciales (monohilo), en paralelo (multihilo) y en arquitecturas distribuidas (sin memoria compartida) [35].

Todas las implementaciones de xDEVS, para sus diferentes versiones en distintos lenguajes de programación, están disponibles en el sitio web de Github[3], una plataforma online que permite que los desarrolladores colaboren y compartan sus proyectos. Además, en cada una de las ramas del repositorio en Github de xDEVS se detallan los procesos de instalación y configuración, incluyéndose también ejemplos para su prueba [25].

2.2. *Benchmarking* y métricas de rendimientos

En el estudio del rendimiento y el comportamiento de los sistemas, el *benchmarking* ocupa un lugar privilegiado. Podemos definir el término *benchmark* como un conjunto estandarizado de herramientas para la evaluación y comparación competitiva de sistemas y componentes de acuerdo a características especificas, como el rendimiento o la seguridad [41].

Por otro lado, al diseñar un *benchmark* particular se busca además que se verifiquen los siguientes principios claves que garantizan un alto nivel de

[2] `https://msdl.uantwerpen.be/git/`
[3] `https://github.com/`

calidad en la evaluación que se realiza [30]:

1. **Relevancia**. Cómo de próximo es el comportamiento del sistema durante el *benchmark* con el comportamiento del sistema en los escenarios habituales sobre los que opera.

2. **Reproducibilidad**. Esto es, la habilidad para producir consistentemente resultados similares cuando se ejecuta con la misma configuración.

3. **Justicia**. Debe permitirse la competencia de diferentes configuraciones del test sin limitaciones artificiales.

4. **Verificabilidad**. Los resultados del *benchmark* deben ser confiables en su exactitud.

5. **Usabilidad**. Se tienen que evitar bloqueos que no permitan que los usuarios ejecuten el *benchmark* en los entornos de prueba deseados.

Algunos ejemplos de *benchmarks* bien conocidos en el campo de la informática son los elaborados por el consorcio sin ánimo de lucro SPEC (del inglés, Standard Performance Evaluation Corporation, Coorporación de Evaluación Estándar del Rendimiento) [8], que ofrecen herramientas para la evaluación de componentes hardware como las CPUs.

2.2.1. Métricas de rendimiento

Durante la evaluación de computadores o sistemas es habitual la búsqueda de un indicador numérico único que permita especificar el rendimiento, siendo este un tema candente de estudio en la comunidad científica [37]. El objetivo es tal que, tras ejecutar un *benchmark*, se produzca como resultado una métrica o estimador del rendimiento que evalúe el desempeño del sistema en las tareas de prueba llevadas a cabo. Dependiendo de la tarea a realizar, será interesante la obtención de diferentes métricas. Desafortunadamente, es habitual que el uso de una u otra métrica puede degenerar en errores de análisis, siendo complicada la elección de un enfoque para sistemas complejos. De hecho, es recurrente encontrar análisis e investigaciones que incurren en este error. Por ello, se han publicado estudios dedicados

exclusivamente al análisis de los datos, para así poder concluir qué métricas proporcionan la óptica adecuada desde la que aproximarse a la evaluación de los resultados obtenidos [37, 19, 30, 18].

A modo de ejemplo podemos destacar uno de los errores más habituales relacionados con el uso equívoco de métricas. Este es, la utilización de la media aritmética sobre conjuntos de valores normalizados (en lugar de emplear la media geométrica) [19]. A modo ilustrativo, supongamos que tenemos tres máquinas M_1, M_2 y M_3 en las que ejecutamos dos *benchmarks* diferentes B_1 y B_2, obteniendo diferentes tiempos de ejecución. Los resultados de la ejecución anterior se recogen en la tabla 2.1.

	M_1	M_2	M_3
B_1	2	1	4
B_2	4	8	2

Cuadro 2.1: Resultados M_1, M_2 y M_3 en los *benchmarks* B_1 y B_2.

De los datos en la tabla se deduce que la máquina M_1 ejecuta las pruebas de B_1 el doble de lenta que M_2, mientras que es el doble de rápida que M_3 para este mismo *benchmark*. Sin embargo, en el caso de B_2 esta relación se invierte, siendo M_2 el doble de lenta que M_1 y este último a su vez el doble que M_3. Una conclusión natural, que se podría derivar de este hecho, es que el comportamiento de las máquinas M_1, M_2 y M_3 es parecido, solo que algunas se comportan mejor en ciertas condiciones respecto a otras. Sin embargo, normalizando respecto M_1 y calculando la media aritmética y geométrica sobre los resultados obtenemos algunos resultados sorprendentes, tabla 2.2.

Podemos observar entonces que tanto M_2 como M_3 presentan una variación del 25 % con respecto a M_1. La conclusión obtenida anteriormente puede inducir a error, pensando que existe una discrepancia del 25 % de rapidez entre la máquina M_1 y las máquinas M_2 y M_3, siendo además estas últimas totalmente equivalentes. Sin embargo, este hecho no se deriva de las observaciones sobre los datos. Por otro lado, notemos también que se ha calculado la media geométrica sobre los valores normalizados. El valor reportado en este caso es equivalente para todas las máquinas. Este hecho es consistente con la idea de que las máquinas se comportaron de manera

	M_1	M_2	M_3
B_1	1.00	0.5	2.00
B_2	1.00	2.00	0.5
Media aritmética	1.00	1.25	1.25
Media geométrica	1.00	1.00	1.00

Cuadro 2.2: Media aritmética y valores normalizados con respecto a M_1.

similar, variando el tiempo de ejecución para compensarse su desempeño al considerar ambos *benchmarks* (cuando una de las máquinas arrojaba resultados de ejecución en la mitad de tiempo que otra para un benchmark, para el contrario su tiempo de ejecución era el doble).

Tal y como ilustra el ejemplo anterior, es muy importante escoger correctamente la métrica para analizar el rendimiento de los sistemas que se quieren evaluar. Una mala elección de las herramientas matemáticas a emplear para el análisis de los resultados puede degenerar, como en el caso anterior, en observaciones poco relevantes, que inducen a confusión o error con respecto a los resultados reales de los tests ejecutados.

2.2.2. Medias pitagóricas

En la subsección anterior destacamos la importancia de utilizar métricas de rendimiento adecuadas para cada caso de estudio, exponiendo el uso de dos de las métricas más habituales como son la media geométrica y la aritmética. En particular, ambas medias pertenecen a una clase de medias más generales que denominamos medias pitagóricas. En esta subsección estudiaremos este tipo de medias, sus propiedades más generales y la idoneidad de estas para caracterizar diferentes conjuntos de datos. Comencemos definiendo formalmente las medias pitagóricas:

Definición 2.2.1. Se denominan medias pitagóricas a las medias aritmética $\overline{x}_{AM}$, geométrica $\overline{x}_{GM}$ y armónica $\overline{x}_{HM}$, tales que para un conjunto de

N elementos, $\{x_0, \ldots, x_{N-1}\}$, se pueden expresar como

$$\overline{x}_{AM} = \frac{\sum_{k=0}^{N-1} x_k}{N}\ , \quad \overline{x}_{GM} = \sqrt[N]{\prod_{k=0}^{N-1} x_k}\ , \quad \text{y} \quad \overline{x}_{HM} = \frac{N}{\sum_{k=0}^{N-1} \frac{1}{x_k}}.$$

Las medias anteriores cumplen una serie de propiedades que las hacen interesantes desde el punto de vista matemático y útiles para su aplicación práctica. Entre ellas, destacamos la homogeneidad de primer orden (esto es, si $\overline{x}$ es una media pitagórica sobre los valores $\{x_0, \ldots, x_{N-1}\}$ entonces la media pitagórica $b{\cdot}\overline{x}$ lo es de los valores $\{b{\cdot}x_0, \ldots, b{\cdot}x_{N-1}\}$), la preservación del valor (si $\{x_0, \ldots, x_{N-1}\}$ verifica que $x_i = a$ para todo elemento x_i del conjunto, entonces $\overline{x} = a$), la invarianza bajo intercambio (esto es, el orden de los elementos del conjunto no varía el valor de la media) y la propiedad de ser promedio (esto es, para un conjunto de valores X se verifica que $\min X \leq \overline{x} \leq \max X$).

Asimismo, incluimos a continuación el siguiente resultado que permite acotar las medias pitagóricas entre sí y con respecto a los elementos del conjunto analizado.

Teorema 2.2.1 (Desigualdad de las medias)**.** Dado un conjunto de N elementos, $X = \{x_0, \ldots, x_{N-1}\}$, se verifica la siguiente desigualdad

$$\min X \leq \overline{x}_{HM} \leq \overline{x}_{GM} \leq \overline{x}_{AM} \leq \max X\ .$$

Una vez estudiadas algunas de las propiedades más básicas sobre las medias pitagóricas, que garantizan su buen comportamiento en el análisis de datos como métricas, podemos proceder a plantearnos cuándo es más útil emplear una frente a la otra. Notemos que, a pesar de haber expuesto las medias más utilizadas en el análisis de datos, existen muchas otras que pueden ser de interés para su revisión en futuros estudios [18]. Es más, matemáticamente, todas las medias expuestas hasta ahora pueden englobarse bajo el concepto de media de Hölder[4].

Algunas reglas generales que se han derivado en la literatura del análisis del rendimiento proponen utilizar las medias aritméticas para la agregación

[4]La media de Hölder, $\overline{x}_H$, o media generalizada para un conjunto $\{x_0, \ldots, x_{N-1}\}$, se

de tiempos y la armónica para proporciones [26], mientras que otras como la expuesta en la subsección anterior sugieren la utilización de la media geométrica para conjuntos de valores normalizados [19]. Sin embargo, en vez de limitarnos a esquemas fijos, estudiaremos las propiedades de las medias pitagóricas sobre los datos, infiriendo a partir de estas su idoneidad para diferentes casos de estudio. En particular, nuestro análisis se basará en una aproximación a través de la Ley de Amdahl [11], que establece que el aumento en el rendimiento debido a una mejora en un cierto componente del sistema está limitado por el tiempo que se utiliza dicho componente en la ejecución de una tarea. Dicho de otra forma, los componentes que más se utilizan son los que más interés tienen al estudiar el rendimiento. Este hecho puede derivarse matemáticamente a través de la expresión formal de la Ley de Amdahl:

$$A = \frac{1}{(1-F) + \frac{F}{x}}$$

donde A es la mejora total, F es la fracción de tiempo de uso del componente y x es el factor de mejora del componente. Así, si simplemente mejoramos indefinidamente un componente (esto es, incrementamos el valor del factor x), obtenemos que

$$\lim_{x\to\infty} A = \lim_{x\to\infty} \frac{1}{(1-F) + \frac{F}{x}} = \frac{1}{1-F}.$$

De manera que la mejora total alcanzada se encuentra acotada y determinada por la fracción de uso del componente.

Por ello, estudiaremos la aplicabilidad de métricas en función de su relación con la mediana[5], permitiéndonos así obtener métricas que no queden

define para cierto $p \in \mathbb{Z}$ como

$$\overline{x}_H = (\frac{1}{N} \sum_{k=0}^{N-1} x_k^p)^{\frac{1}{p}},$$

de manera que si $p = -1$ se obtiene la media armónica, con $p = 0$ la geométrica y con $p = 1$ la aritmética.

[5]La mediana representa el valor de la variable de posición central en un conjunto de datos ordenados, de manera que el 50 % de los valores son menores o iguales que este y el otro 50 % son mayores o iguales.

degeneradas por valores en las colas superiores o inferiores de la distribución, esto es, valores muy grandes o muy pequeños que distorsionen el estimador.

Estudio de la media aritmética

La media aritmética se computa para un conjunto de datos sumando todos ellos y dividiendo entre el número de elementos. De esta manera, la relación que se establece entre los términos es lineal y de equitatividad entre los valores. Este hecho conlleva diferentes consecuencias en la aplicación de la media aritmética. En particular, dado que la relación entre los términos está modelada por un comportamiento lineal, en general la media aritmética es adecuada para la agregación de datos que siguen un comportamiento lineal. El caso más conocido donde se verifica este hecho es en el cálculo de la esperanza matemática, pero existen otros. Supongamos que tenemos el conjunto de valores $X_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, entonces la media aritmética $\overline{x}_{AM} = 5$ denota precisamente el valor que, escogiendo números aleatorios del conjunto, es más probable que esté lo más cercano posible a estos valores. Esto es, la media aritmética aproxima de forma certera la mediana para el conjunto anterior, la figura 2.5 ilustra este hecho.

Figura 2.5: Representación del conjunto X_1 y el valor de la media aritmética coincidente con la mediana

En relaciones lineales, podemos decir entonces que la media aritmética es equitativa para los términos. Sin embargo, en el caso de relaciones no lineales, la media aritmética no suele arrojar resultados igual de esclarecedores. Por el contrario, supongamos ahora que nos dan el conjunto $X_2 = \{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$ y computemos la media aritmética $\overline{x}_{AM} = 204.6$, siendo la mediana 48. Claramente el valor de la media se encuentra desplazado notablemente respecto al valor de la mediana. La figura 2.6 ilustra este escenario.

Esto implica que, para casos de no linealidad, la media aritmética no captura el comportamiento de los datos de forma realista, favoreciendo a

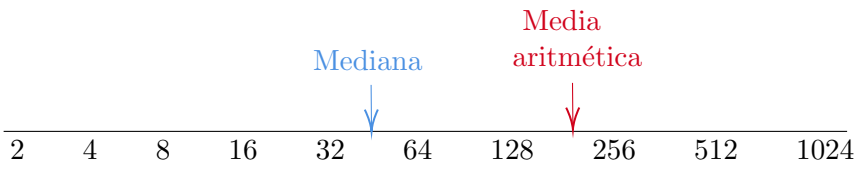


Figura 2.6: Representación del conjunto X_2, su media aritmética y mediana.

los valores más grandes. Esto coincide con la idea que puede derivarse del Teorema de la Desigualdad de las Medias, en el que se puede comprobar que la media aritmética suele sobrestimar los valores frente a los otros tipos de medias, armónica o geométrica.

A su vez, notemos también que por la operación empleada para agregar los valores en la media aritmética, la suma de los términos, se preservan las unidades de los valores. Esto es, si tenemos un conjunto de valores dados en la unidad u, por ejemplo de la forma $\{1u,\ 2u,\ 3u\}$, la media de los valores se expresa también en la unidad u, $\overline{x}_{AM} = 2u$.

Por último, es importante destacar que, tal y como se cubrió en la sección 2.2.1, la media aritmética no es adecuada para conjuntos de datos normalizados ya que puede verse alterada drásticamente en función de la escala de referencia [19].

Estudio de la media geométrica

Mientras que en el caso de la media aritmética observamos un buen comportamiento sobre conjuntos de datos cuya interrelación se puede modelar linealmente, en el caso de las medias geométricas la relación que se da entre los términos es multiplicativa, no lineal. Por la misma forma de agregar los datos, realizando el producto de todos ellos, la media geométrica proporciona un valor que se aproxima mejor a la mediana en poblaciones de datos con comportamientos no lineales. Veamos este hecho con el ejemplo anterior propuesto en el estudio de la media aritmética, esto es, con el conjunto $X_2 = \{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$. Para este conjunto, se obtiene que $\overline{x}_{GM} = 45.25$, que es notablemente más cercano al valor de la mediana, 48, frente a la media aritmética. La figura 2.7 ilustra la semejanzas con la mediana y la discrepancias entre ambos tipos de medias.

Claramente, el conjunto X_2 presenta un comportamiento no lineal.

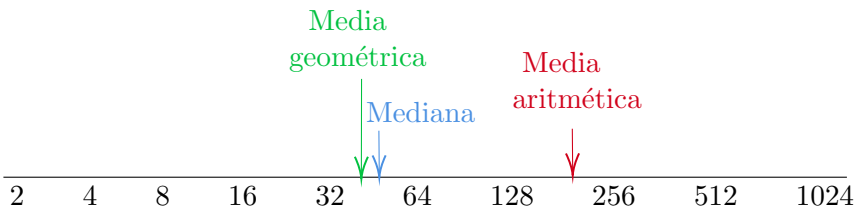


Figura 2.7: Representación del conjunto X_2, su media aritmética, geométrica y mediana.

Nótese que cada elemento x_i puede obtenerse como $x_i = 2 \cdot x_{i-1}$. Así, queda patente que la media geométrica, al tratar de representar los datos de forma que la media aproxime estadísticamente los datos con certeza, parece más adecuada para el modelizado de datos con comportamientos no lineales frente a la media aritmética. En general, para este tipo de conjuntos de datos la media geométrica no discrimina valores bajos, frente a la media aritmética que favorece excesivamente los valores grandes.

Sin embargo, al contrario que en el caso de la media aritmética, la media geométrica no preserva correctamente las unidades. Es más, a diferencia de la aritmética, la media geométrica puede ser usada con independencia de la escala de referencia empleada al normalizar un conjunto de datos. De esta manera, la media geométrica es adecuada para conjuntos normalizados [19]. Este hecho puede derivarse a partir de la siguiente propiedad que cumple la media geométrica. Denotemos X_i a una secuencia de longitud n y a Y_i a otra secuencia de valores de igual longitud, entonces si $GM(\cdot)$ denota la media geométrica de un conjunto se verifica que

$$GM(\frac{X_i}{Y_i}) = \frac{GM(X_i)}{GM(Y_i)}.$$

La media geométrica es la única de las medias pitagóricas que cumple la relación anterior. Por tanto, es la única que permite trabajar con valores normalizados de forma independiente a la escala. En el caso de las métricas de rendimiento, en las que se suele escoger un sistema de referencia, el uso de la media geométrica es de importancia capital, al permitir no discriminar ciertos sistemas frente a otros en función de la escala de referencia escogida. Sin embargo, el uso de la media geométrica ha sido cuestionado en

algunos estudios [37], bajo el lema de que proporcionar resultados consistentes no siempre equivale a proporcionar resultados correctos (además de romper con la escala de las unidades de los valores analizados en el resultado, mientras que otras medias como la aritmética la preservan). De hecho, generalmente, las disparidades arrojadas por el uso de la media armónica o la aritmética en función del sistema de referencia pueden explicarse por los diferentes pesos que se asigna a cada *benchmark*, de manera que utilizando medias pitagóricas ponderadas podría reducirse el impacto asignando pesos relacionados con el coste real de cada *benchmark*, normalizando posteriormente los resultados con respecto a alguno de los sistemas una vez computada la media ponderada.

Finalmente, y para concluir, notemos que la media geométrica se anula completamente en conjuntos donde existe al menos un valor nulo, de manera que no permite trabajar con poblaciones de datos donde alguno de los elementos puede anularse.

Estudio de la media armónica

La media armónica puede expresarse como el recíproco[6] de la media aritmética de los recíprocos de un conjunto. De hecho, la media armónica presenta propiedades análogas a las de la media aritmética, pero invirtiendo su efecto. Por ejemplo, mientras que la media aritmética favorece a los valores más grandes de un conjunto, la media armónica favorece a los más pequeños. A modo de ejemplo, tomemos X_2 definido en los subapartados anteriores, cuya media armónica es $\overline{x}_{HM} = 10.01$. Este valor dista notablemente tanto de la mediana, como la media geométrica o la aritmética. En general, sobrestimando a los valores inferiores de la distribución. La figura 2.8 ilustra este hecho.

Por otro lado, la media armónica se presenta de especial interés a la hora de trabajar con razones o proporciones. Supongamos, a modo de ejemplo, que queremos viajar de una ciudad C_1 a otra ciudad C_2 y volver. Ambas ciudades distan la una de la otra $200km$. Durante el trayecto de ida, viajamos a una velocidad constante de $v_1 = 100km/h$, mientras que a la vuelta, debido al tráfico, la velocidad se mantiene constante en $v_2 = 50km/h$. Si tomásemos la media aritmética de ambos valores obtendríamos que la

[6] El recíproco, o inverso, de un elemento $x \neq 0$ es $\frac{1}{x}$.

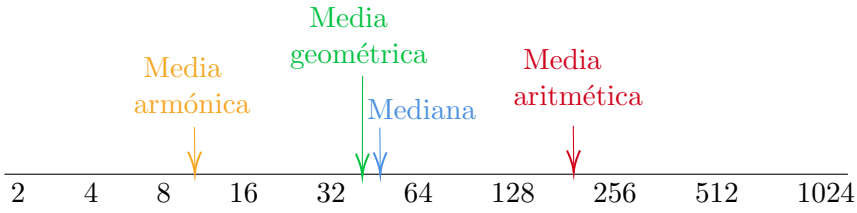


Figura 2.8: Representación del conjunto X_2, su media aritmética, geométrica, armónica y mediana.

velocidad media estimada es de $75km/h$, y si tomamos la geométrica obtendríamos $70km/h$. Sin embargo, notemos que, mientras que en la ida viajamos a $100km/h$ durante $2h$, la vuelta a $50km/h$ duró $4h$. Esto es, durante un porción mayor del tiempo la velocidad fue sustancialmente menor. De hecho, si tomamos la media armónica para estos valores obtendremos que la velocidad media es de $\sim 66km/h$. En este caso podríamos preguntarnos si la estimación de las diferentes medias es o no la adecuada para determinar una velocidad media del viaje, teniendo en cuenta la diferencia de tiempo invertida en cada trayecto. Una forma de proceder podría ser ponderar el viaje en función del tiempo consumido en cada trayecto. Así, podemos computar la media aritmética ponderada del viaje total en relación al tiempo dedicado a cada sentido de la marcha, de manera que se obtiene que la velocidad media es

$$\overline{v} = \frac{2h \times 100km/h + 4h \times 50km/h}{2h + 4h} \approx 66km/h$$

justamente una aproximación de la media armónica expuesta. Y es que, en general, la media armónica sirve para aproximar de forma adecuada proporciones.

Así, avalando los resultados hasta ahora expuestos en referencia al uso de la media armónica en problemas involucrando velocidades y tramos, la conclusión habitual que puede encontrarse en la literatura es que la media armónica se debe utilizar siempre cuando existen períodos o relaciones de proporcionalidad entre los datos [26].

Finalmente, podemos destacar que, al igual que ocurría con la media aritmética, la media armónica no se comporta de forma adecuada con conjuntos de datos normalizados, al ser dependiente de la escala. Además, al

igual que ocurría con el caso de la media geométrica, esta media no arroja resultados coherentes en el caso de que alguno de los elementos del conjunto de datos a analizar sea nulo.

2.2.3. Medias ponderadas

En ocasiones es interesante no sólo el estudio de las medias pitagóricas, sino también de su versión ponderada. La ponderación de medias consiste en asignar pesos a cada uno de los valores posibles del conjunto del que queremos computar alguna métrica. Así, pasamos a definir las medias pitagóricas ponderadas como sigue:

Definición 2.2.2. Se denominan medias pitagóricas ponderadas, o simplemente medias ponderadas, a las medias aritmética ponderada $\overline{x}_{WA}$, geométrica ponderada $\overline{x}_{WG}$ y armónica ponderada $\overline{x}_{WH}$, tales que para un conjunto de N elementos, $\{x_0, \ldots, x_{N-1}\}$, con N valores no negativos $\{w_0, \ldots, w_{N-1}\}$, que denominaremos pesos, se pueden expresar como

$$\overline{x}_{WA} = \frac{\sum_{k=0}^{N-1} x_k w_k}{\sum_{k=0}^{N-1} w_k}, \quad \overline{x}_{WG} = \left(\prod_{k=0}^{N-1} x_k^{w_k} \right)^{\frac{1}{\sum_{k=0}^{N-1} w_k}}, \quad \text{y} \quad \overline{x}_{WH} = \frac{\sum_{k=0}^{N-1} w_k}{\sum_{k=0}^{N-1} \frac{w_k}{x_k}}.$$

En particular, cuando los pesos verifican que $\sum_{k=0}^{N-1} w_k = 1$, se dice que están normalizados, y pueden simplificarse las expresiones anteriores. A modo de ejemplo, para el caso anterior, la media aritmética ponderada se puede expresar como $\overline{x}_{WA} = \sum_{k=0}^{N-1} x_k w_k$. Notemos que en general siempre es posible normalizar los pesos, basta con realizar la siguiente transformación sobre los pesos originales $w'_j = \frac{w_j}{\sum_{k=0}^{N-1} w_k}$.

El uso de las medias ponderadas es particularmente útil en casos en los que existe disparidad de importancia o semántica entre los valores a analizar. En la subsección anterior se introdujo ya, como ejemplo, el caso de un viaje de ida y vuelta entre dos ciudades con distintas velocidades en cada trayecto. En dicha situación la media aritmética no justifica de forma fiel la velocidad media real del viaje, pero ponderando con respecto al tiempo invertido en cada tramo se puede obtener un resultado más certero.

Otros ejemplos de medias ponderadas, dentro de la misma industria de la medición del rendimiento, se encuentran en la métrica utilizada para el *benchmark* SPECviewperf ® [7], un estándar utilizado para estimar el rendimiento gráfico en aplicaciones de uso profesional. En este caso, se utiliza como métrica la media geométrica ponderada, con pesos dispuestos en relación con el porcentaje de tiempo dedicado a que cada uno de los conjuntos de prueba fuesen visualizados en los tests.

2.2.4. DEVStone, un *benchmark* para DEVS

Con el objetivo de poder evaluar con un *benchmark* las simulaciones basadas en DEVS surge DEVStone, un *benchmark* sintético que aspira a poder automatizar completamente este tipo de evaluaciones. DEVStone facilita el análisis de rendimiento para versiones sucesivas de un mismo motor de simulación, y proporciona un modelo para poder comparar rendimiento en diferentes entornos de modelado y simulación. Este *benchmark* puede verse como un generador de modelos sintéticos que produce modelos diversos con estructuras y comportamientos que mezclan operaciones comunes [21], de manera que facilite la ejecución de pruebas del tipo deseado. En general, DEVStone se centra en producir modelos en función de diferentes parámetros, como son:

- El **tipo** del modelo. Esto es, las diferentes estructuras y esquemas de interconexión entre componentes.
- La **profundidad**, d. Es decir, el número de niveles en la jerarquía de modelos.
- La **anchura**, w. Esto es, el número de componentes en cada modelo acoplado intermedio empleado.
- El **tiempo de transición interno**. Es decir, el tiempo de ejecución invertido en cada función de transición interna.
- El **tiempo de ejecución externo**. Esto es, el tiempo invertido en las funciones de transición externas.

Cada uno de los modelos queda completamente especificado a través de estos parámetros.

Por otro lado, y de acuerdo a las especificaciones de DEVStone [21], para construir cada modelo se utilizan modelos acoplados artificiales con la profundidad especificada d de componentes acoplados. Todos y cada uno de ellos consisten en un total de $w - 1$ modelos atómicos (con la excepción del nivel inferior de la jerarquía, que está compuesto por un único modelo atómico). Además, las transiciones internas o externas funcionan ejecutándose en un porcentaje de tiempo determinado, con el *benchmark* sintético Dhrystone [46] para mantener el consumo de ciclos del reloj del procesador[7].

En este trabajo nos centraremos en cuatro de los modelos principales introducidos hasta ahora en la literatura [44], sin embargo es importante destacar también que nuevos modelos están siendo estudiados y propuestos [35]. Los modelos en los que nos centraremos son los siguientes:

- **LI**. Modelos que tienen pocas interconexiones para cada modelo acoplado.
- **HI**. Modelos con un alto grado de acoplamiento en las entradas.
- **HO**. Modelos con un alto nivel de acoplamiento y numerosas salidas.
- **HOmod**. Modelos con un nivel exponencial de acoplamiento y salidas.

Cada uno de estos modelos puede ser utilizado con diferentes propósitos, dependiendo del sistema que se quiera evaluar. La principal métrica obtenida, que será de especial interés en futuros capítulos, es el tiempo de ejecución total. En el caso de herramientas de simulación basadas en mensajes (como es el caso de los sistemas DEVS), también puede almacenarse el tipo y número de mensajes para realizar análisis internos de la simulación [44] (así como para comprobar la correcta ejecución del modelo). También, en el caso de querer realizar pruebas de esfuerzo, se pueden generar modelos de gran tamaño, pudiéndose medir la utilización de memoria a través del aumento iterativo del tamaño de los distintos modelos ejecutados.

En el resto de esta subsección pasamos a describir cada uno de los modelos anteriormente expuestos de forma pormenorizada.

[7]Notemos que Dhrystone se compone de un conjunto reducido de instrucciones de aritmética entera, por lo que es un candidato ideal para analizar modelos en DEVS basados en la utilización de variables de estado discretas.

Modelo de baja interconectividad, LI

La estructura general de los modelos LI se basa en construir $d-1$ capas (donde $d \geq 1$ es la profundidad del modelo). Cada una de estas capas tiene un modelo acoplado y $w-1$ modelos atómicos (donde $w \geq 2$ es la anchura del modelo). La figura 2.9 representa cada una de las capas intermedias de un modelo LI.

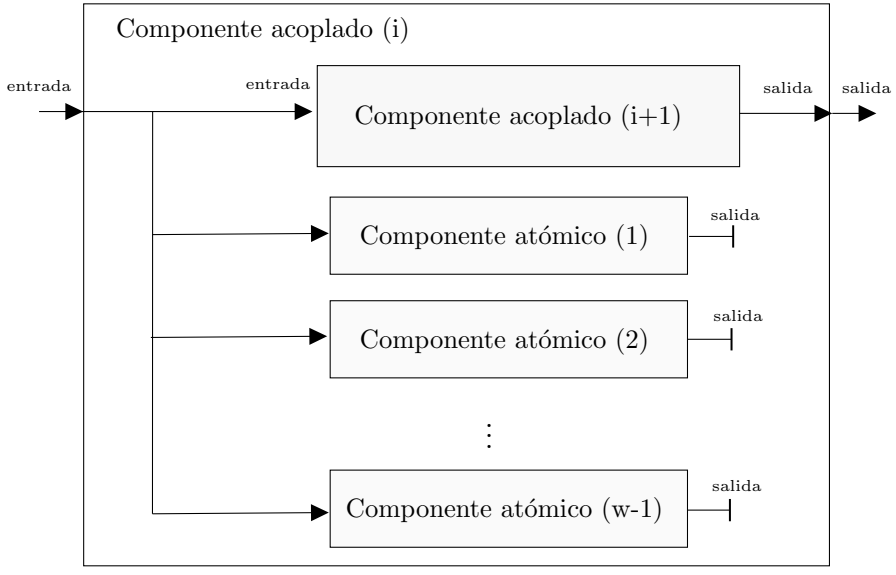


Figura 2.9: Representación de las capas intermedias i-ésimas del modelo LI.

Por otro lado, la última capa difiere del resto y puede representarse tal y como se muestra en la figura 2.10. Notemos que en este caso la capa solo contiene un único modelo atómico con el que se procesarán eventos.

Por último, podemos destacar e inferir a partir de las representaciones anteriores que los modelos LI producen una transición externa ($\#\delta_{ext}$), un evento de salida ($\#eventos$) y una transición interna ($\#\delta_{int}$) por cada modelo atómico interno y evento externo inducido. Así, para los modelos LI el número de transiciones y eventos generados es igual al número de modelos atómicos multiplicado por el total de eventos inducidos N. En general trabajaremos siempre con $N = 1$, de forma que se verifican las

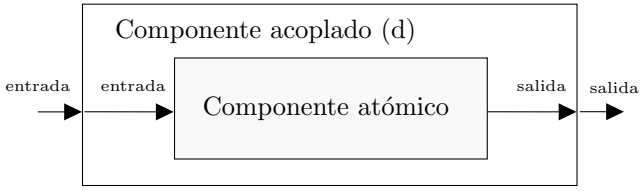


Figura 2.10: Representación de la capa más profunda del modelo LI.

siguientes ecuaciones [35]:

$$\#\delta_{int} = ((w-1)\cdot(d-1)+1), \tag{2.1}$$

$$\#\delta_{ext} = ((w-1)\cdot(d-1)+1), \tag{2.2}$$

$$\#eventos = ((w-1)\cdot(d-1)+1). \tag{2.3}$$

Modelo de alta interconectividad, HI

El modelo de alta interconectividad HI se construye de forma semejante a como se hizo con LI, sin embargo para este caso el puerto de salida de cada componente atómico se conecta con el puerto de entrada del siguiente componente atómico. La figura 2.11 muestra esta estructura.

Por construcción, el número de componentes atómicas es el mismo que el que se tenía en el modelo LI. Sin embargo, las funciones de transición y los eventos generados difieren a causa de la conexión entre los puertos de salida y entrada. De hecho, la entrada externa actúa sobre los modelos atómicos como un registro de desplazamiento, generando un evento adicional por cada uno externo. De esta manera, se puede recomputar el total de eventos y transiciones de forma análoga a como se hizo en el caso de LI, obteniéndose [35]:

$$\#\delta_{int} = \left(\frac{w^2-w}{2}\right)\cdot(d-1)+1, \tag{2.4}$$

$$\#\delta_{ext} = \left(\frac{w^2-w}{2}\right)\cdot(d-1)+1, \tag{2.5}$$

$$\#eventos = \left(\frac{w^2-w}{2}\right)\cdot(d-1)+1. \tag{2.6}$$

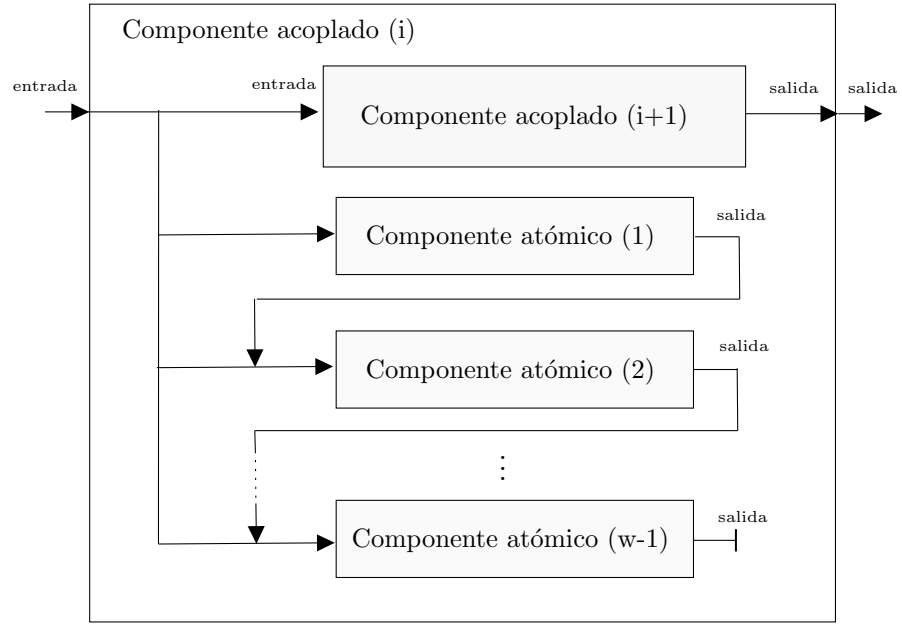


Figura 2.11: Representación las capas intermedias del modelo HI

Modelo HI con múltiples salidas, HO

El modelo HO se caracteriza por ser un modelo que presenta un mapa de interconexiones más complejo, con el mismo número de unidades atómicas y acopladas que presentaba HI. En particular, las unidades acopladas en HO poseen dos puertos de entrada y dos de salida en cada nivel. La principal diferencia con HI radica precisamente en los nuevos puertos, estando el segundo puerto de entrada de cada componente acoplada conectado a la entrada de cada modelo atómico. Además, la salida de cada modelo atómico está conectada a la segunda salida de su modelo acoplado padre. La figura 2.12 ilustra esta relación.

A partir de la estructura descrita, se puede derivar que el número de componentes atómicas, funciones de transición y eventos generados en HO coinciden con los del modelo HI. Sin embargo, la diferencia en tiempos de ejecución y la huella en memoria es notable, y se debe a la presencia de mayores puertos de entrada externos. Así, se pueden modelar las ecuaciones

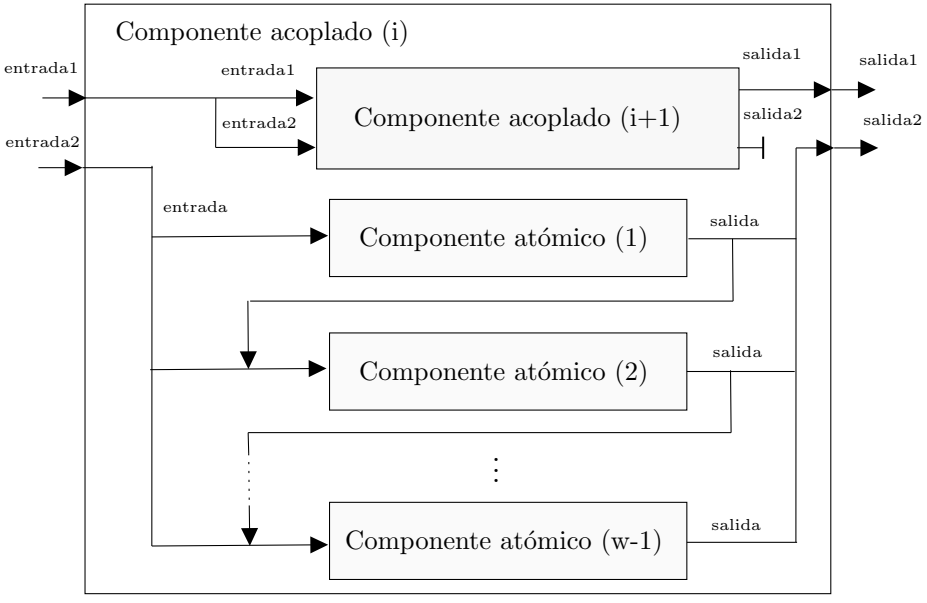


Figura 2.12: Representación las capas intermedias del modelo HO

que describen el total de eventos e interacciones del modelo como [35]:

$$\#\delta_{int} = \left((w-1) + \sum_{k=1}^{w-2} k \right) \cdot (d-1) + 1, \tag{2.7}$$

$$\#\delta_{ext} = \left((w-1) + \sum_{k=1}^{w-2} k \right) \cdot (d-1) + 1, \tag{2.8}$$

$$\#eventos = \left((w-1) + \sum_{k=1}^{w-2} k \right) \cdot (d-1) + 1. \tag{2.9}$$

Modelo HOmod

Los modelos HOmod buscan incrementar el tráfico de mensajes, siendo este aumento exponencial en el intercambio de mensajes entre modelos acoplados. Para ello, se utiliza un conjunto de $w-1$ modelos donde cada

uno de los componentes atómicos activa un conjunto de $w-1$ modelos atómicos. Estos tienen sus salidas conectadas a la segunda entrada del modelo acoplado en cada nivel. De esta manera, los modelos reciben una cantidad de eventos relacionados de forma exponencial con la profundidad y la anchura de cada nivel. Los eventos externos son retransmitidos por cada componente acoplado a las componentes atómicas y acopladas hijas, y el proceso se repite para cada módulo acoplado hasta que se llega a un componente hoja. La figura 2.13 muestra claramente la arquitectura de los modelos HOmod de forma genérica.

El cálculo de las ecuaciones que modelan las transiciones en este caso es bastante más complejo, y su desarrollo detallado puede encontrarse en [35].

Recogemos a continuación únicamente las expresiones para $\#\delta_{int}$ y $\#\delta_{ext}$, cuya forma es más simple y será de interés en próximas secciones:

$$\#\delta_{int} = (d-1)(w-1)^2 + \left((d-1) + (w-1)\sum_{k=1}^{d-2} k\right) \times \left((w-1) + \sum_{k=1}^{w-1} k\right) + 1 \tag{2.10}$$

$$\#\delta_{ext} = (d-1)(w-1)^2 + \left((d-1) + (w-1)\sum_{k=1}^{d-2} k\right) \times \left((w-1) + \sum_{k=1}^{w-1} k\right) + 1 \tag{2.11}$$

2.2.5. Complejidad en los modelos DEVStone

Pasaremos en esta subsección a estudiar la complejidad, en función de la anchura y profundidad, de los principales modelos DEVStone introducidos anteriormente. En este *benchmark* lo fundamental es considerar las ejecuciones de Dhrystone para el consumo de CPU. Así, pasaremos a estudiar la complejidad de cada modelo en función del número de transiciones internas o externas, esto es, en relación con los valores de las funciones δ_{ext} y δ_{int}.

Para denotar cada una de las complejidades de cada modelo m en función de su anchura w y profundidad d, escribiremos $O_m(w,d)$. Esta expresión se computará siempre como $O_m(w,d) = O(\text{máx}\{\#\delta_{int}, \#\delta_{ext}\})$ donde

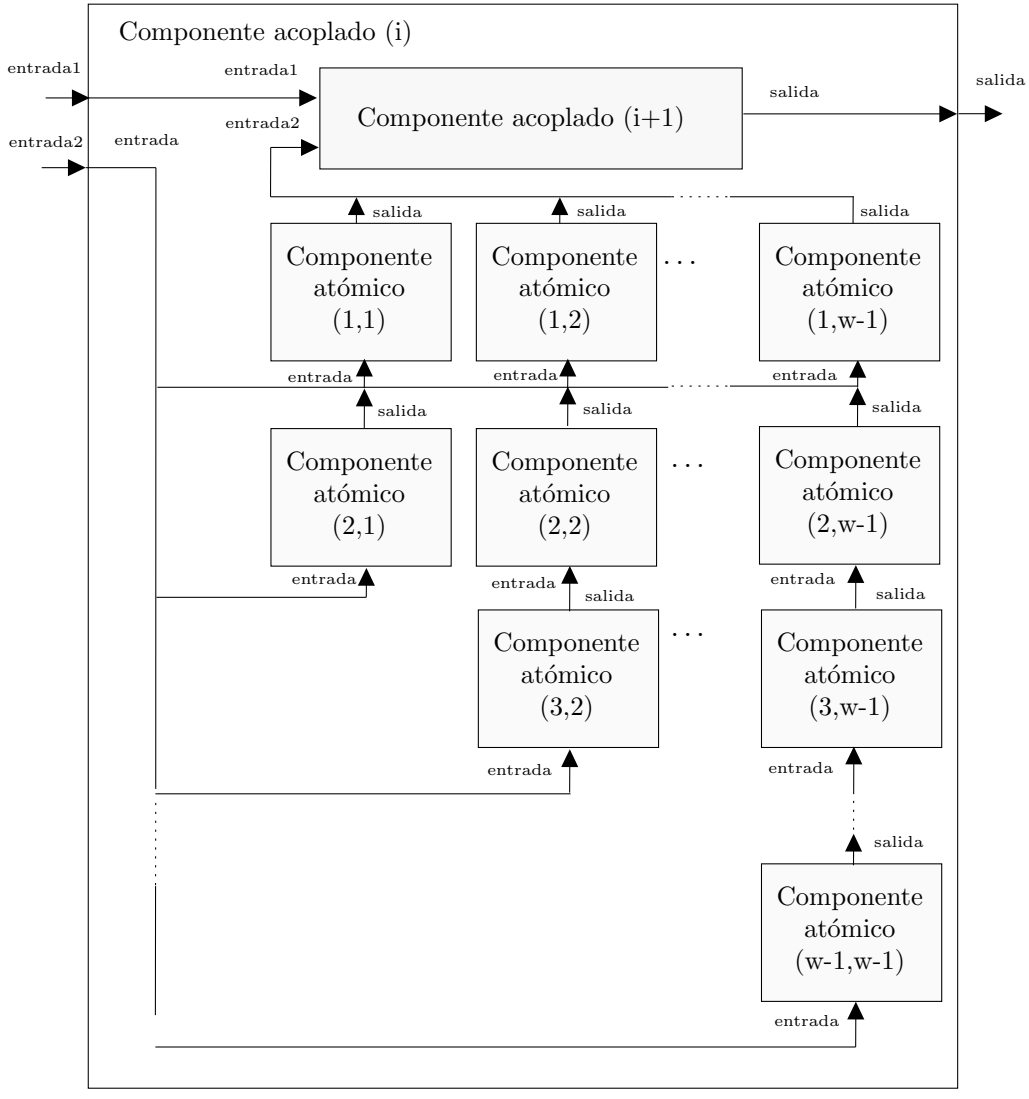


Figura 2.13: Representación las capas intermedias del modelo HOmod

la operación $O(\cdot)$, en el lado derecho de la igualdad, indica la notación de Landau para el estudio de las complejidades. En futuros capítulos, hacien-

do un abuso de notación, realizaremos la siguiente identificación de forma indistinta para operar $O_m(w,d) = O(g(w,d)) = g(w,d)$, donde $g(w,d)$ es una cierta función que recoge la complejidad del modelo m.

Cuando exista la necesidad de distinguir, además del modelo y del tipo de simulador ts empleado, se escribirá la complejidad como $O_m^{ts}(w,d)$.

Complejidad del modelo LI

Es inmediato que, a partir de la ecuaciones (2.1) y (2.2), la complejidad de los modelos LI depende del producto $w \times d$, ya que

$$O((w-1)\cdot(d-1)+1) = O(w\cdot d),$$

de forma que, podemos decir que la complejidad del modelo LI es $O_{LI}(w,d) = w \cdot d$.

Complejidad del modelo HI

Podemos proceder con HI de forma análoga a como se estudió el caso del modelo LI. Así, observando las ecuaciones (2.4) y (2.5), podemos derivar la complejidad como

$$O\left(\left(\frac{w^2-w}{2}\right)\cdot(d-1)+1\right) = O(w^2\cdot d).$$

De esta manera, podemos decir que la complejidad del modelo HI es $O_{HI}(w,d) = w^2 \cdot d$.

Complejidad del modelo HO

Recordemos que, como ya se expuso en la subsección 2.2.4, tanto HO como HI presentan el mismo número de funciones de transición. Así, claramente, se tiene que $O_{HI}(w,d) = O_{HO}(w,d) = w^2 \cdot d$.

Complejidad del modelo HOmod

Finalmente, observando las ecuaciones (2.10) y (2.11), podemos estudiar la complejidad del modelo HOmod en relación con las transiciones y

en función de la anchura y profundidad. Para ello, primero recordemos que, en general, se verifica que

$$\sum_{k=1}^{N-2} k = \frac{1}{2}(N-2)(N-1) \quad \text{y} \quad \sum_{k=1}^{N-1} k = \frac{1}{2}(N-1)N.$$

Haciendo uso de las anteriores igualdades podemos desarrollar los términos de la ecuación (2.10), pudiendo entonces computar la complejidad como

$$O\left((d-1)(w-1)^2 + \left((d-1) + (w-1)\sum_{k=1}^{d-2} k\right) \times \left((w-1) + \sum_{k=1}^{w-1} k\right) + 1\right)$$

que es equivalente a

$$O\left((d-1)(w-1)^2 + [(d-1) + (w-1)\frac{1}{2}(d-2)(d-1)] \times [(w-1) + \frac{1}{2}(w-1)w]\right),$$

siendo esta complejidad

$$O(\text{máx}\{dw^2, w^3d^2\}) = O(w^3d^2).$$

De manera que obtenemos que $O_{HOmod}(w, d) = w^3d^2$.

Capítulo 3

Metodología

En este capítulo describiremos la metodología empleada en los experimentos para medir el rendimiento de los simuladores de eventos discretos. Para ello, comenzaremos describiendo el marco de trabajo general sobre el que operamos. Esto es, cubriremos el entorno de trabajo y ejecución donde se realizaron las pruebas, así como los tipos de simuladores empleados y sus características principales.

Tras esto, pasaremos a describir el enfoque utilizado en cuanto a la evaluación y análisis de los resultados obtenidos. Primero, plantearemos la estrategia utilizada para evaluar los resultados, discriminando entre ejecuciones erróneas y pudiendo estudiar los fallos que pudieran producirse en cada instanciación. Seguidamente, expondremos las métricas propuestas para el análisis de los resultados de ejecución de los simuladores para cada modelo, así como los posibles métodos que deben compararse para el análisis del rendimiento global del simulador.

3.1. Marco de trabajo

Durante nuestro estudio, nos centramos en buscar métricas de utilidad que permitan estimar los resultados de tiempos de ejecución obtenidos por el *benchmark* DEVStone. Esta *suite* de pruebas permite la generación sintética de diferentes modelos parametrizables, facilitando el cómputo de simulaciones representativas que puedan ilustrar el rendimiento del sistema

a analizar.

En esta sección pasaremos a describir tanto el entorno de ejecución en el que se han realizado las pruebas, como los distintos modelos y simulaciones que se han llevado a cabo. De esta manera, buscamos detallar el marco de trabajo, favoreciendo la replicabilidad de los resultados obtenidos y exponiendo los detalles técnicos que determinaron las mediciones.

3.1.1. Entorno de ejecución

Para realizar las ejecuciones de los diferentes simuladores se optó por utilizar Google Cloud[1], un espacio virtual a través del cual se habilita la utilización de diversos servicios de almacenamiento, procesado y gestión de datos. En particular, el servicio empleado para el desarrollo de las pruebas y simulaciones fue Compute Engine[2]. Este servicio ofrece la posibilidad de configurar entornos de máquinas virtuales con los que trabajar en la nube.

Entre la familia de máquinas disponibles para su uso se escogió N1, una familia de máquinas de propósito general que ofrece CPUs virtuales (vCPU)[3] sobre las microarquitecturas Intel Sandy Bridge, Ivy Bridge, Haswell, Broadwell y Skylake. La fecuencia base de los procesadores anteriores oscila levemente entre los 2.0 y los 2.6 GHz.

Para nuestro estudio se desplegó una máquina virtual con una CPU del tipo Intel Haswell, con 1 vCPU y 3.75 GB de memoria. En concreto, dentro de la plataforma de cloud de Google, este tipo de dispositivos reciben el nombre de *n1-standard-1*[4]. La tabla 3.1 recoge los detalles técnicos principales hasta ahora expuestos de la máquina a utilizar.

El sistema operativo instalado en la máquina empleada para la ejecución de las simulaciones fue Debian [3], a partir de la imagen debian-10-buster-v20210701.

[1] Se puede encontrar más información en su página web: `https://cloud.google.com/`.

[2] Más información disponible en: `https://cloud.google.com/compute`.

[3] Una vCPU representa una porción compartida de una CPU física asignada a una cierta máquina virtual. En general, su uso está extendido en servicios de computación en la nube donde los procesadores que se ofrecen implementan técnicas de *hyperthreading*, permitiendo la ejecución de varios hilos en un mismo núcleo.

[4] Este nombre puede descomponerse, de forma genérica, como *familia-propósito-número de vCPUs*.

Tipo de máquina	CPUs virtuales	Memoria (GB)	Microarquitectura
n1-standard-1	1	3.75	Intel Haswell

Cuadro 3.1: Tipo de máquina virtual para las simulaciones.

Notemos entonces que este dispositivo ejecuta una distribución de Linux, de manera que podemos obtener más información de la CPU de la máquina virtual utilizada a través de la información almacenada en el sistema de archivos[5]. Así, puede ejecutarse el siguiente comando para obtener más información sobre la CPU:

```
$ cat /proc/cpuinfo
```

De esta manera, se obtuvieron los detalles que se recogen en la figura 3.1 para la máquina empleada.

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU @ 2.30GHz
stepping        : 0
microcode       : 0x1
cpu MHz         : 2299.998
cache size      : 46080 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2
 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmu
lqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm invpcid
_single pti ssbd ibrs ibpb stibp fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt arat md_clear arch_c
apabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs
bogomips        : 4599.99
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

Figura 3.1: Detalles técnicos de la CPU de la máquina virtual.

Para la ejecución de las pruebas se utilizó el sistema de gestión de tareas

[5]Desde directorio */proc* se permite el acceso a un pseudo sistema de ficheros que contiene información de los datos en la jerarquía de ficheros especiales que representan el estado actual del *kernel*. Esta información se detalla dentro del propio manual de referencia de Linux [5].

y de clústeres Slurm (del inglés, Simple Linux Utility for Resource Management, Utilidad Simple de Linux para la Gestión de Recursos) [6]. A través de Slurm pueden enviarse simulaciones en forma de trabajos a ejecutar en las máquinas del clúster. Para ello, se utilizó la siguiente configuración para cada tarea:

```
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
```

Además, para las ejecuciones de simulaciones de modelos en Java se reservó memoria extra que garantizase una correcta ejecución sin excepciones. Para alcanzar este objetivo, se añadió la siguiente opción en el comando java:

```
$ java -Xmx4g
```

3.1.2. Simuladores y modelos

Siendo nuestro principal objetivo estudiar el rendimiento de diferentes simuladores de eventos discretos, escogimos entre los simuladores disponibles un subconjunto representativo. Estos simuladores debían ser representativos tanto en arquitectura y diseño, como en el lenguaje de programación empleado en su implementación. De esta manera, buscamos aquellas versiones desarrolladas en los lenguajes comúnmente más empleados hoy en día, como son Java, Python[6] o C++. Las implementaciones de DEVS seleccionadas para los simuladores fueron aDEVS, PyPDEVS y xDEVS (véase la subsección 2.1.3 para más información). La tabla 3.2 detalla los tipos de simuladores empleados, la versión y el lenguaje de implementación de los mismos.

Por otro lado, para el estudio del desempeño de los diferentes simuladores con el *benchmark* DEVStone, se seleccionaron cuatro de los principales modelos disponibles: LI, HI, HO y HOmod. Estos modelos, introducidos en la sección 2.2.4, quedan completamente determinados al fijar una anchura w y una profundidad d. En general, a mayor profundidad o anchura el coste computacional se incrementa, en ocasiones de forma exponencial (por ejemplo en el caso de los modelos HOmod). Así, el enfoque tomado fue

[6]Durante este trabajo nos referiremos indistintamente a la implementación CPython y a la versión del lenguaje Python3 simplemente como Python.

Implementación	Lenguaje	Versión
aDEVS	C++	3.3
xDEVS	C++	1.0.0
xDEVS	Java	1.1.0
PythonPDEVS	Python3	2.4.1

Cuadro 3.2: Detalles de los simuladores analizados.

ejecutar diferentes instancias de los mencionados cuatro modelos, variando los parámetros w y d, para cada uno de los simuladores a analizar.

De esta manera, para los modelos LI, HI y HO se computaron instancias con w y d variando de 100 a 2000 con variaciones de 50 entre los términos. Esto es $w, d \in \{z \in \mathbb{Z}^+ \ : \ z = 100 + 50 \cdot n \ \text{ y } n \in \{0, 1, \ldots, 38\}\}$. Por otro lado, y de forma totalmente análoga, para el caso del modelo HOmod se utilizaron instancias de forma que $w \in \{z \in \mathbb{Z}^+ \ : \ z = 2 + n \ \text{ y } n \in \{0, 1, \ldots, 8\}\}$ y $d \in \{z \in \mathbb{Z}^+ \ : \ z = 1 + n \ \text{ y } n \in \{0, 1, \ldots, 9\}\}$. La diferenciación de las ejecuciones entre HOmod y los otros modelos se debe a la complejidad exponencial que exhiben, dificultando la computación de instancias parametrizadas con valores grandes de profundidad o anchura. A su vez, es destacable que en el caso de HOmod no tiene sentido la definición de $w = 1$ pero sí de $d = 1$, por ello las instancias probadas empezaron a partir de $w = 2$.

La elección de los valores de profundidad y tamaño para los diferentes modelos se obtuvo de forma empírica. Aunque se recojan en la metodología de trabajo, la elección de los valores fue consecuencia directa de la experimentación, que arrojó detalles sobre las limitaciones de nuestro sistema para ejecutar modelos complejos. Estas limitaciones no eran siempre por el tiempo, sino también por incapacidad física en cuanto a memoria disponible en las máquinas.

En total, se realizaron 1521 ejecuciones de cada modelo HO, HI y LI. Además, también se realizaron 90 ejecuciones de HOmod. Notemos que este cálculo es por cada simulador, de manera que en total para cada uno se obtuvieron 1611 ejecuciones por cada simulador, que agregadas dan un total de $1611 \cdot 4 = 6444$ ejecuciones.

3.2. Corrección, análisis y evaluación

Cubriremos en esta sección el enfoque tomado para estudiar los resultados, detectando posibles errores, y las estrategias propuestas para el análisis del rendimiento global de los simuladores, así como el desempeño local en cada modelo probado. Además, incluiremos una sección donde proponemos diferentes aproximaciones a la evaluación de las métricas utilizadas en el análisis de los resultados.

Los resultados obtenidos para el análisis y evaluación se derivan de ejecuciones de simulaciones de diferentes modelos por parte de los distintos simuladores. Cada ejecución realizada reportaba dos ficheros distintos, un fichero con extensión *.out* que recogía los resultados y otro *.err* detallando posibles errores. En el caso de existir algún error, el fichero *.out* se creaba vacío. En caso contrario, en ausencia de error, el fichero *.err* era el que se generaba sin contenido. Cada simulador gestionaba la ejecución de simulaciones para cada una de las anchuras, profundidades y tipos de modelos especificados en la subsección 3.1.2. De esta manera, se producían tantos ficheros de salida y error como instancias de simulación se planearon. Para el análisis de los resultados de ejecución y error se crearon diferentes scripts en Python. Estos compilaban los detalles más relevantes en ficheros *.csv* de resultados y *.txt* de resumen de errores. La figura 3.2 ilustra este proceso.

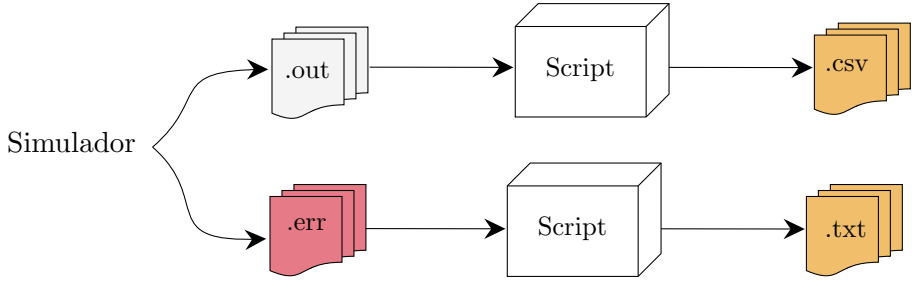


Figura 3.2: Proceso de obtención de los ficheros de evaluación y resultados.

En las siguientes subsecciones cubriremos un estudio previo de la corrección de las ejecuciones, exponiendo los posibles errores así como las formas de proceder en caso de que estos se diesen. Notemos que en el análisis de errores también es posible encontrar diferencias y comparar así los

simuladores entre sí (si un simulador empieza a producir errores en instancias grandes por problemas de memoria y otro no, esto quiere decir que el segundo realiza una gestión más eficiente de los recursos). También, veremos la generación de resultados a partir de los ficheros de salida creados por el simulador, y explicaremos las métricas propuestas para el análisis de las ejecuciones de los diferentes modelos. Además, propondremos distintas métricas para un análisis del rendimiento global, comentando la relevancia y aplicabilidad de las mismas. Finalmente, expondremos las estrategias utilizadas para la evaluación de las métricas y los resultados obtenidos.

3.2.1. Corrección de las ejecuciones

Para comprobar la correcta ejecución y determinar posibles errores en las simulaciones, dada la gran cantidad de ficheros a analizar (véase la subsección 3.1.2), se optó por escribir un *script* en Python que iterase sobre los ficheros *.err* generados en busca de fallos. Este *script* no sólo anotaba los modelos que producían errores, sino también la anchura, profundidad y causa del fallo. De esta manera, se compilaba un fichero de resumen de error para cada simulador y modelo, pudiéndose así identificar fácilmente posibles problemas de configuración o ejecución de las instancias.

Entre los potenciales errores contemplados, a priori, podemos listar los siguientes:

- Error de memoria (*Memory error*). Cuando se busca reservar memoria para un modelo acoplado o atómico habiendo agotado la memoria disponible en el computador.

- Desbordamiento de pila (*Stack overflow*). En el caso de que la pila reservada para el programa en el entorno de ejecución de la aplicación no sea suficiente para las llamadas anidadas a funciones.

- Código distinto de cero de salida al terminar un trabajo. En general, Slurm está configurado para recibir como señal la salida de terminación de un trabajo. Potencialmente alguna implementación de los simuladores podría omitir el código de salida nulo en ejecuciones correctas.

Notemos que los errores anteriores suelen generar excepciones que, generalmente, no pueden gestionarse por el sistema operativo o el intérprete que alojan el entorno de ejecución. Por ello, es habitual la interrupción abrupta de la simulación en estos casos, impidiendo el volcado de los resultados en los ficheros *.out*. A modo de ejemplo, este es el caso de los problemas de memoria con Python. En dicha situación, dado que, en las implementaciones en CPython, la arquitectura de gestión de la memoria subyacente se realiza a través de operaciones `malloc()`, en ocasiones el intérprete no puede recuperarse del puntero nulo devuelto al no poder reservar espacio para la creación de un objeto[7]. En el caso de los errores de desbordamiento de pila, estos se dan, por ejemplo en el caso de Java, cuando el tamaño de memoria requerido por la pila del programa es mayor de lo que previamente *Java Runtime Environment* reservó para la aplicación.

En resumen, la mayoría de errores que asumimos que podían ocurrir se debían a la limitación de recursos (*Memory error* o *Stack overflow*) o bien a pequeñas modificaciones salvables fácilmente en las implementaciones de los simuladores analizados. En el caso de los errores debidos a limitaciones, optamos por tratar de repetir las ejecuciones un par de veces más. Si no había éxito en las repeticiones, simplemente pasaríamos a considerar que el modelo era demasiado complejo para ser ejecutado con los recursos disponibles y obviaríamos la instancia concreta en el análisis. Esta estrategia nos permitió, además, conocer los límites alcanzados por el *hardware* para generar la población de datos.

3.2.2. Métricas para el análisis de los modelos

Los simuladores arrojan como resultados para cada modelo los tiempos de creación del modelo t_{ms}, de configuración t_c y el tiempo de simulación t_s. De esta manera, el tiempo total o tiempo de ejecución de la simulación puede computarse como

$$t = t_{ms} + t_c + t_s.$$

Notemos entonces que por cada modelo m, profundidad d y anchura w se produce un tiempo total de ejecución que expresaremos como $t^m_{(w,d)}$. Así,

[7] Respecto a esta cuestión pueden encontrarse más detalles en la documentación oficial del lenguaje, `https://docs.python.org/3/library/exceptions.html#exceptions.MemoryError`.

por ejemplo, para el modelo HO de profundidad 50 y anchura 100 se tiene que el tiempo total es $t^{HO}_{(100,50)}$.

En consideración con lo anterior, es posible obtener un conjunto de tiempos de simulación para cada modelo. La agregación de tiempos entre modelos presenta una mayor complejidad que la agregación total de tiempos dentro del mismo modelo, pero ambas cuestiones requieren un análisis previo. A priori, y en función de lo expuesto en la sección 2.2, al trabajar con tiempos, la aproximación más adecuada sería a través de la utilización de la media aritmética. Sin embargo, notemos que la media aritmética funcionaría de forma adecuada si repitiésemos ejecuciones para un mismo modelo, a una misma profundidad y anchura. Esto es, cómputos de la forma

$$\overline{t^m_{(w,d)}} = \frac{1}{N}\sum_{k=0}^{N-1} t^m_{(w,d),\,k}$$

son adecuados para estimar agregaciones de elementos de un mismo tipo, donde se quiere obtener una aproximación media del total de datos. En el escenario planteado, dado un conjunto de anchuras W y profundidades D para un modelo m, buscamos reducir a un solo elemento $\overline{t^m}$ la dimensionalidad del conjunto

$$\{t^m_{(w,d)} \ : \ w \in W, d \in D\}.$$

Propuesta de métrica para el análisis de modelos

Nuestra propuesta original, que surge de la intuición teórica recogida en las secciones anteriores (2.2.1 y 2.2.2), se basará en computar la media geométrica ponderada sobre los tiempos obtenidos. La media geométrica es adecuada para esta tarea por ser invariante con respecto al sistema de referencia, y permitir la comparación posterior entre modelos de diferentes simuladores. De hecho, la media geométrica se comporta mejor que la media aritmética en tareas donde existe posible normalización de datos o comparación posterior utilizando proporciones [31]. Así, la métrica que modelice el desempeño total para un cierto modelo, $\overline{t^m}$, será un reflejo independiente de la escala a usar y permitirá establecer razones de proporción de mejor o peor rendimiento.

Por otro lado, la función de ponderación será el orden de complejidad estimado que cada modelo presenta con respecto a la anchura y profundidad. De esta manera, se explicita la complejidad añadida por cada modelo mediante profundidad y anchura, espaciando la muestra de datos acorde a la complejidad teórica real de cómputo. Los detalles referidos a la complejidad pueden encontrarse en la subsección 2.2.5.

Con todo, es importante destacar que, en ocasiones, no es posible realizar el cómputo de la media geométrica ponderada utilizando la expresión habitual y bien conocida

$$\overline{x}_{GM} = \left(\prod_{k=0}^{N-1} x_k^{w_k} \right)^{\frac{1}{\sum_{k=0}^{N-1} w_k}}.$$

Esta situación ocurre cuando los valores evaluados x_i cumplen que $x_i < 1$ o x_i es un valor elevado, siendo N un número grande. La problemática anterior se ve potenciada en los casos en que combinemos productos de números en diferentes órdenes de magnitud (muy grandes y muy pequeños). En este escenario, es difícil operar computacionalmente el número $\prod_{k=0}^{N-1} x_k$ en coma flotante. Este hecho se ve acentuado por la elevación de cada término x_i a una potencia $w_i > 1$. Es más, en caso de que la precisión en aritmética de coma flotante sea suficientemente amplia como para cubrir la expresión decimal del número anterior (incluso tras la aplicación de la operación potencia con respecto de los pesos), la operación exponente sobre un número cercano al cero o demasiado grande suele producir excepciones. Esto se debe a que para valores muy grandes de N y $\sum_{k=0}^{N-1} w_k$ se verifica que $\overline{x}_{GM} \approx 0^0$ o $\overline{x}_{GM} \approx M^{M'} \approx +\infty$, con M y M' grandes.

En su lugar, y para evitar los problemas anteriores, expresaremos la media geométrica de la siguiente forma equivalente

$$\overline{x}_{GM} = exp\left(\frac{\sum_{k=0}^{N-1} w_k \ln x_k}{\sum_{k=0}^{N-1} w_k} \right).$$

De esta manera, las magnitudes operadas quedan mejor dimensionadas. Notemos que hemos cambiado la base de la exponencial de un número

cercano a 0, o un número muy grande, por la base de la exponencial natural. Además, el exponente está formado por sumas del logaritmo natural, que escalará los números pequeños (y los más grandes) para que sea más fácil operar sobre ellos[8].

Consecuentemente, fijado un conjunto de anchuras W y profundidades D, y dada una complejidad $O_m(w,d)$ podemos expresar la métrica obtenida para el análisis de cada modelo m como

$$\overline{t^m} = exp\left(\frac{\sum\limits_{w\in W,\, d\in D} O_m(w,d)\ln x_k}{\sum\limits_{w\in W,\, d\in D} O_m(w,d)}\right). \tag{3.1}$$

Recordemos que, $O_{LI}(w,d) = wd$, $O_{HI}(w,d) = O_{HO}(w,d) = w^2d$ y $O_{HOmod}(w,d) = w^3d^2$. Así, queda completamente caracterizada la métrica $\overline{t^m}$ para cada uno de los modelos estudiados.

Propiedades de la métrica $\overline{t^m}$

Nuestro candidato a métrica (3.1) verifica una serie de propiedades que lo hacen adecuado para el estudio de rendimientos. Notemos primero que uno de nuestros objetivos era poder comparar el desempeño de un simulador al ejecutar un determinado modelo. De esta manera, podemos denotar la métrica anterior para un simulador s y un modelo m como $\overline{t^{m,s}}$. Así, supongamos que tenemos tres simuladores distintos s_1, s_2 y s_3, para los que queremos comparar su rendimiento con respecto un cierto modelo m. Para lograr este objetivo, es posible escoger un sistema de referencia, por ejemplo el simulador s_1, para que midiendo la proporción con este puedan compararse los simuladores, esto es,

$$c_1 = \frac{\overline{t^{m,s_1}}}{\overline{t^{m,s_2}}} \quad \text{y} \quad c_2 = \frac{\overline{t^{m,s_1}}}{\overline{t^{m,s_3}}}.$$

De esta manera, cuando $c_1 > 1$ diremos que s_2 es generalmente mejor que s_1 al simular el modelo m, y para valores de $c_1 < 1$ diremos que es peor. De

[8]La aproximación teórica planteada se encuentra amparada en las pruebas realizadas. Durante el análisis de los datos se trató de utilizar primero la expresión comúnmente conocida de la media geométrica ponderada, arrojando esta errores y excepciones. Es por esto que optamos por proceder tal y como se ha expuesto.

forma análoga puede procederse con c_2. Es más, comparando los valores de c_1 y c_2, pueden también obtenerse las relaciones entre s_2 y s_3. Si $c_1 > c_2$ entonces s_2 presenta un mejor desempeño en las ejecuciones del modelo m, y para $c_2 > c_1$ sucede lo contrario. Notemos que, por construcción de la métrica $\overline{t^m}$, estas comparativas pueden realizarse con independencia de la máquina de referencia elegida. Esto es, si en vez de s_1 escogiésemos s_2 y computásemos c'_1 y c'_2, obtendríamos que ambos valores muestran los mismos resultados coherentemente con respecto a los anteriores valores. A saber, si para c_1 y c_2 pudimos deducir que s_1 era mejor que s_2, y que s_2 era mejor que s_3, estas conclusiones se tendrán igualmente computando y analizando c'_1 y c'_2. Este hecho se deriva naturalmente de la invarianza de la normalización en la media geométrica.

Por otro lado, el valor $\overline{t^m}$ obtenido no representa una magnitud física, como pudiera ser el tiempo medio de ejecución para el modelo. El valor presentado puede medirse como el rendimiento genérico, obteniendo una magnitud al normalizarse con respecto a un sistema de referencia. En dicho caso, podemos hablar de la unidad de los valores como c_1 o c_2 en términos de rendimiento con respecto al sistema s_1.

Finalmente, $\overline{t^m}$ presenta algunas ventajas claras con respecto a otro tipo de métricas utilizando medias aritméticas, armónicas, o la geométrica no ponderada. Con respecto a la aritmética podemos destacar la incoherencia del resultado aportado. El valor aportado por esta media resulta tener una magnitud física que realmente no valora el tiempo medio de ejecución de un modelo. Es más, este tiempo medio dependerá de la profundidad y anchura ejecutadas, siendo esta complejidad no lineal (véase la subsección 2.2.5), siendo el comportamiento no lineal no adecuado para el uso de esta métrica (y sí aplicándose al caso de la media geométrica, que se comporta mejor en esquemas multiplicativos como el que se tiene en función de las complejidades para los distintos modelos). Al hecho anterior, podemos sumarle el mal comportamiento de la media aritmética, y también de la armónica, con respecto a la normalización y comparativa utilizando proporciones. Por último, la media geométrica no ponderada presenta un problema de mal escalado con respecto a las complejidades de los modelos. La utilización de los pesos en la media ponderada para este caso busca explicitar que, para instancias más complejas donde la profundidad y anchura es mayor, es proporcionalmente más importante el tiempo de ejecución. En

estas instancias límites los recursos disponibles se aprovechan al máximo para poder resolver las simulaciones, siendo estos valores considerados en mayor medida.

3.2.3. Análisis del rendimiento global

De forma similar a como se estudió la aplicación de métricas en la subsección anterior, en este caso cubriremos otra propuesta de métrica pero para el análisis del rendimiento global. En este escenario, no contemplamos la agregación de datos referidos a tiempos de ejecución de un mismo simulador y modelo para diferentes instancias de profundidad y anchura. En su lugar, cubriremos la aproximación a una compilación de datos referidos a las métricas anteriores estudiadas para la ejecución de modelos. Así, nuestra población de datos estará formada precisamente por estimadores $\overline{t^{m,s}}$ para cada modelo m, fijado un simulador s. Denotaremos $\overline{r_s}$ al valor total que agregue, para un simulador s, todos los estimadores de cada modelo. Nuestro objetivo será que esta métrica cuantifique fiel y coherentemente el rendimiento global del simulador.

En este caso, nuestra propuesta de métrica idónea para la agregación de los datos de la forma $\overline{t^{m,s}}$ es la media geométrica no ponderada, normalizada con respecto a las ejecuciones equivalentes del simulador aDEVS. Así, podemos medir los valores $\overline{r_s}$ para cada simulador s sobre un conjunto de modelos M como

$$\overline{r_s} = \sqrt[(\#M)]{\prod_{m \in M} \overline{t^{m,s}}}, \tag{3.2}$$

donde $\#M$ denota el cardinal de M. Normalizando el valor anterior con respecto del valor $\overline{r_{aDEVS}}$ se puede obtener una métrica del rendimiento global que toma como unidad de referencia la proporción de rendimiento con respecto a aDEVS. Así, se obtiene como métrica de rendimiento global de un sistema s el valor

$$r_s = \frac{\overline{r_{aDEVS}}}{\overline{r_s}}. \tag{3.3}$$

El valor r_s recoge el rendimiento global del sistema s considerados un conjunto de modelos M. Es importante destacar que, para el correcto cómputo de todos los valores anteriores, los modelos M evaluados deben

arrojar valores $\overline{t^{m,s}}$ de forma que estos se computen para los mismos conjuntos de profundidades y pesos. Esto es, en realidad el valor r_s depende también del conjunto de anchuras y profundidades empleado en las simulaciones de cada modelo.

Propiedades de $\overline{r_s}$

La principal ventaja de utilizar la métrica $\overline{r_s}$ para la medición del rendimiento global radica en el uso de la media geométrica para computarla. De esta manera, r_s, aunque utiliza aDEVS para estandarizar los datos, puede computarse de forma equivalente para otros sistemas de referencia con resultados coherentes. Este hecho se deriva naturalmente de la invarianza de la media geométrica con respecto a la normalización de los valores. Así, la media geométrica se muestra más adecuada que la armónica o la aritmética.

Es más, la media armónica no sería correcta por su poca relación con proporciones entre los estimadores de los diferentes modelos. Estos valores representan una agregación del rendimiento y no una tasa de variación o proporcionalidad. Un hecho parecido se da con respecto a la media aritmética. La media aritmética pondera de forma equitativa y lineal los modelos, sin embargo esta interpretación es incorrecta por constituir elementos de diferente índole.

Finalmente, la cuestión de ponderar los valores en el caso de las métricas $\overline{t^{m,s}}$ no presenta ventaja alguna. De escoger algún sistema de pesos este sería aparentemente artificial, y serviría únicamente para sobrestimar el impacto de algún modelo sobre el resto.

3.2.4. Evaluación

Para la implementación de las métricas, y el estudio y evaluación de su comportamiento sobre los datos, se programó un *notebook* en el entorno de programación interactivo Google Colab [4]. En dicho *notebook* se implementaron los cálculos de las métricas para los diferentes modelos utilizando las librerías de Pandas [32], Numpy [22] y SciPy [28]. Además, se usó la librería Matplotlib [24] para representar gráficamente los resultados obtenidos durante la evaluación.

Así, en el *notebook* se utilizaron mapas de calor, que permitieron visualizar claramente el coste en tiempo de ejecución de cada modelo y simulador, en función de la anchura y la profundidad. Los mapas de calor son una técnica de visualización de datos que permite calibrar la magnitud de una variable en colores utilizando solo dos dimensiones. Esto es, permiten medir el impacto de una variable en función de otras dos. La variación del color puede darse en estos casos en tono o intensidad.

Un análisis cualitativo de los mapas de calor puede permitirnos estudiar la adecuación de las métricas empleadas para derivar conclusiones sobre los datos.

Finalmente, expondremos gráficas de barras comparativas de las diferentes métricas sobre los modelos, para así poder ilustrar las relaciones y conclusiones que pueden inferirse a partir de los valores numéricos obtenidos.

Capítulo 4

Resultados

En este capítulo recogeremos los resultados obtenidos tras la ejecución de los simuladores siguiendo el esquema de trabajo del capítulo 3.

Así, estudiaremos primero la corrección en la ejecución de las simulaciones, revisando los errores encontrados y exponiendo la forma de proceder con los datos para las instancias fallidas. Tras esto, explicaremos los mapas de calor relativos a cada uno de los modelos y simuladores, comentando las características principales de cada uno. A su vez, realizaremos un análisis cualitativo de las ejecuciones, señalando las diferencias más significativas entre modelos y simuladores.

Finalmente, aplicaremos las métricas cubiertas en las subsecciones 3.2.2 y 3.2.3 a los resultados de las simulaciones ejecutadas para los diferentes modelos y simuladores. Además, compareremos las diferentes aproximaciones a través del uso de las distintas medias pitagóricas como estimadores. También, cotejaremos los resultados obtenidos por estos métodos con el análisis cualitativo de las ejecuciones visualizando los mapas de calor.

4.1. Resultados de ejecución

Como se detalló en la subsección 3.1.2, se ejecutaron un total de 6.444 simulaciones, muchas de ellas con tiempos de ejecución del orden de minutos. Así, es común encontrar errores como los contemplados ya en el apartado de metodología (subsección 3.2.1). En esta sección pasaremos a cubrir pri-

mero los errores encontrados en las ejecuciones, analizando la corrección y proponiendo alternativas en los casos en los que no pudo computarse con éxito la simulación.

Seguidamente, pasaremos a visualizar los resultados de la ejecución empleando mapas de calor. Evaluaremos y analizaremos estos mapas para derivar conclusiones cualitativas sobre las ejecuciones, los diferentes modelos y simuladores.

4.1.1. Corrección de las ejecuciones

Tras ejecutar los diferentes simuladores encontramos varios errores recurrentes para instancias de modelos grandes. Destacamos los siguientes:

- **std::bad_alloc**. Error recurrente en el simulador aDEVS. Esta excepción es arrojada en C++ cuando se intenta ejecutar una función de asignación y reserva de memoria sin memoria disponible, por ejemplo, invocando *new*.

- **malloc(), memory corruption (fast)**. Error esporádico en casos límites de instancias de modelos con gran profundidad y anchura, en el simulador aDEVS. Este error se debe al uso de la función *malloc* en vez de emplear *new*. Por ello, no se genera excepción del tipo *bad_alloc* como se esperaría. Este error está relacionado con la implementación de aDEVS escogida, que relega parte de su cometido a librerías escritas en C, pudiendo producirse este tipo de excepciones. Al igual que en el caso anterior, este error está generado por la incapacidad de reservar el tamaño de memoria deseado para instancias de gran anchura o profundidad.

- **MemoryError**. Esta es la excepción más común en PyPDEVS, surgiendo sobre todo en instancias del modelo HOmod y para tamaños grandes de profundidad y anchura. En Python, esta excepción es arrojada cuando el proceso se queda sin memoria disponible para almacenar nuevos datos.

- **Slurm memory exceeded**. En ocasiones, las ejecuciones en Slurm terminaron abruptamente debido al exceso de memoria requerido por el proceso. En estos casos es la propia herramienta de gestión Slurm

la que terminaba con la simulación al colapsar los recursos *hardware* disponibles. Este error era frecuente en las ejecuciones de xDEVS, para las implementaciones en Java y C++.

- **java.lang.StackOverflowError**. Este error apareció en las simulaciones en xDEVS, en su implementación en Java. Este error se debe al uso de algún procedimiento recursivo que realiza excesivas llamadas a la pila.

Los errores anteriores se corresponden precisamente con las situaciones límites derivadas de los dispositivos *hardware* empleados. La tabla 4.1 recoge las simulaciones que fallaron, divididas por simulador.

Modelo	Simulaciones fallidas[1]	
LI	—	
HI	1600x1950 a 1600x2000	1650x1900 a 1650x2000
	1700x1850 a 1700x2000	1750x1850 a 1750x2000
	1800x1750 a 1800x2000	1850x1700 a 1850x2000
	1900x1650 a 1900x2000	1950x1550 a 1950x2000
	2000x1450 a 2000x2000	
HO	1550x2000	1600x1950
	1700x1850 a 1700x2000	1750x1800 a 1750x2000
	1800x1750 a 1800x2000	1850x1700 a 1850x2000
	1900x1650 a 1900x2000	1950x1500 a 1950x2000
	2000x1450 a 2000x2000	
HOmod	2x6 a 2x7	2x9 a 2x10
	4x3	4x5 a 4x6
	4x9 a 4x10	5x7 a 5x10
	6x6 a 6x10	7x6 a 7x10
	8x4	8x6 a 8x10
	9x3 a 9x4	9x6 a 9x10
	10x4 a 10x10	

Cuadro 4.1: Errores en las simulaciones de aDevs.

Modelo	Simulaciones fallidas	
LI	1550x1950 a 1550x2000	1600x1900 a 1600x2000
	1650x1850 a 1650x2000	1700x1800 a 1700x2000
	1750x1750 a 1750x2000	1750x1750 a 1750x2000
	1800x1650 a 1800x2000	1850x850
	1850x1200 a 1850x1250	1850x1500
	1850x1650 a 1850x2000	1900x850
	1900x1150 a 1900x1200	1900x1550 a 1900x2000
	1950x800	1950x1100 a 1950x1200
	1950x1550 a 1950x2000	2000x1100 a 2000x1200
	2000x1550 a 2000x2000	
HI	850x1950 a 850x2000	900x1900 a 900x2000
	950x1800 a 950x2000	1000x1700 a 1000x2000
	1050x1650 a 1050x2000	1100x1600 a 1100x2000
	1150x1500 a 1150x2000	1200x1450 a 1200x2000
	1250x1350 a 1250x2000	1300x1400 a 1300x2000
	1350x1250 a 1350x2000	1400x1300 a 1400x2000
	1450x1250 a 1450x2000	1500x1150 a 1500x2000
	1550x1150 a 1550x2000	1600x1100 a 1600x2000
	1650x1100 a 1650x2000	1700x1050 a 1700x2000
	1750x1000 a 1750x2000	1800x1000 a 1800x2000
	1850x950 a 1850x2000	1900x750
	1900x950 a 1900x2000	1950x900 a 1950x2000
	2000x900 a 2000x2000	
HO	700x2000	750x1900 a 750x2000
	800x1750 a 800x2000	850x1650 a 850x2000
	900x1550 a 900x2000	950x1450 a 950x2000
	1000x1450 a 1000x2000	1050x1350 a 1050x2000
	1050x1350 a 1050x2000	1100x1300 a 1100x2000
	1150x1250 a 1150x2000	1200x1200 a 1200x2000
	1250x1150 a 1250x2000	1300x1100 a 1300x2000
	1350x1050 a 1350x2000	1400x1000 a 1400x2000
	1450x1050 a 1450x2000	1500x950 a 1500x2000

[1]Para la descripción de las simulaciones fallidas se utiliza la notación *anchura* × *profundidad*, de forma que 100x100 indica una ejecución en anchura 100 y profundidad 100.

	1550x900	1550x1000 a 1550x2000
	1600x900 a 1600x2000	1650x850 a 1650x2000
	1700x900 a 1700x2000	1750x800 a 1750x2000
	1800x800 a 1800x2000	1850x800 a 1850x2000
	1900x750 a 1900x2000	1950x750 a 1950x2000
	2000x750 a 2000x2000	
HOmod	6x7 a 6x10	7x6 a 7x10
	8x5 a 8x10	9x4 a 9x10
	10x4 a 10x10	

Cuadro 4.2: Errores en las simulaciones de xDevs Java.

Modelo	Simulaciones fallidas	
LI	—	
HI	1700x2000	1750x1950 a 1750x2000
	1800x1900 a 1800x2000	1850x1850 a 1850x2000
	1900x1750 a 1900x2000	1950x1700
	1950x1800 a 1950x2000	2000x1650
	2000x1750 a 2000x2000	
HO	1400x2000	1450x1950
	1500x1900	1500x2000
	1550x1950	1600x1950 a 1600x2000
	1650x1800 a 1650x2000	1700x1800
	1700x1900 a 1700x2000	1750x1750 a 1750x2000
	1800x1700 a 1800x2000	1850x1650 a 1850x2000
	1900x1600 a 1900x2000	1950x1550 a 1950x2000
	2000x1500 a 2000x2000	
HOmod	4x9 a 4x10	5x8 a 5x10
	6x7 a 6x10	7x6 a 7x10
	8x6 a 8x10	9x6 a 9x10
	10x6 a 10x10	

Cuadro 4.3: Errores en las simulaciones de xDevs C++.

Es importante destacar que las simulaciones que terminaron de forma

abrupta nos permiten también comparar los simuladores entre sí, en función de su capacidad de gestión de los recursos y, en particular, de la memoria. Así, es claro que PypDevs gestiona de forma deficiente la ejecución de modelos pequeños en instancias grandes de profundidad y anchura. De las 4653 simulaciones llevadas a cabo, 3803 fueron fallidas. La mayoría de estos fallos se produjeron en ejecuciones de modelos simples, como LI. Por otro lado, la gestión de PypDevs de memoria y recursos para la simulación del modelo HOmod es excepcionalmente buena, en comparación con aDevs o xDevs que arrojaron multitud de instancias fallidas. Notemos que este modelo es exponencialmente más complejo en cuanto a interconectividad que el resto, de manera que PypDevs parece gestionar de forma adecuada modelos mucho más complejos en conectividad.

Por otro lado, podemos destacar que no hubo simulaciones erróneas de aDevs al ejecutar el modelo LI. De hecho, esto será posteriormente de interés al destacar este simulador en el rendimiento óptimo para la ejecución de modelos simples. Por ello, aparentemente aDevs parece más adecuado para la simulación de instancias simples y no complejas.

También, es notable que las ejecuciones erróneas totales de aDevs y xDevs C++ son parecidas, siendo para el primero 154 y para el segundo 125. En secciones posteriores veremos que, de hecho, el comportamiento de aDevs es similar al de xDevs C++ en multitud de simulaciones analizadas. Por otro lado, las ejecuciones erróneas totales de xDevs Java fueron 950, siendo estas bastante superiores a las de xDevs C++ o aDevs. Al concluir este capítulo analizando el rendimiento global veremos que de hecho xDevs Java ofrece peores resultados, en términos generales, al simular las instancias propuestas de modelos, anchuras y profundidades.

Finalmente, para poder realizar un análisis coherente de todos los simuladores, se asignará el valor máximo de tiempo de ejecución para cada modelo y simulador a las instancias fallidas.

4.1.2. Mapas de calor

Los mapas de calor son un gráfico de representación de datos que permite mostrar resultados de mediciones utilizando distintos colores. Así, los mapas de calor permiten ilustrar la relación de intensidad de una tercera variable en función de otras dos, utilizando para ello un degradado de colores.

En nuestro caso, compararemos el tiempo total de simulación con respecto a la anchura y profundidad de una instancia para un cierto modelo.

Para la representación de los resultados empleando mapas de calor utilizamos una escala de colores común a cada modelo para permitir la comparación entre simuladores. Además, en consideración con lo expuesto en la sección anterior (donde vimos que hubo multitud de ejecuciones fallidas por incapacidad del simulador de soportar modelos de alta complejidad con anchura y profundidad elevadas) decidimos asignar el valor de tiempo total máximo a las ejecuciones fallidas, destacando así claramente que en estas situaciones se requeriría un uso intenso de los recursos con un alto coste, simplificando la representación de los datos y su comparación.

Los mapas de calor son más fáciles de analizar contrastando ejecuciones entre simuladores de un mismo modelo. Así, las figuras 4.1, 4.2, 4.3 y 4.4 representan los resultados en forma de mapa de calor para los modelos LI, HI, HO y HOmod respectivamente.

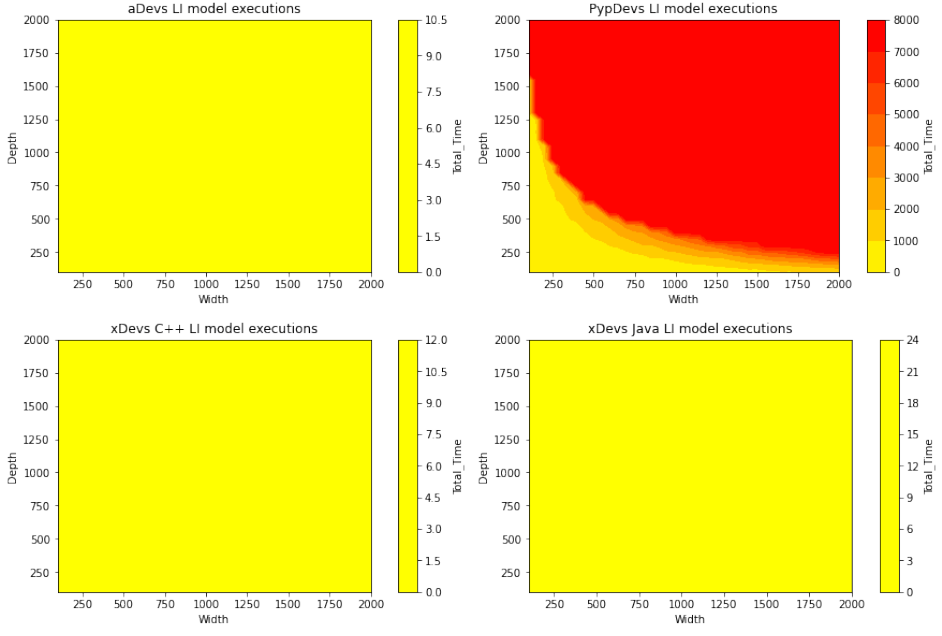


Figura 4.1: Mapas de calor para las ejecuciones del modelo LI.

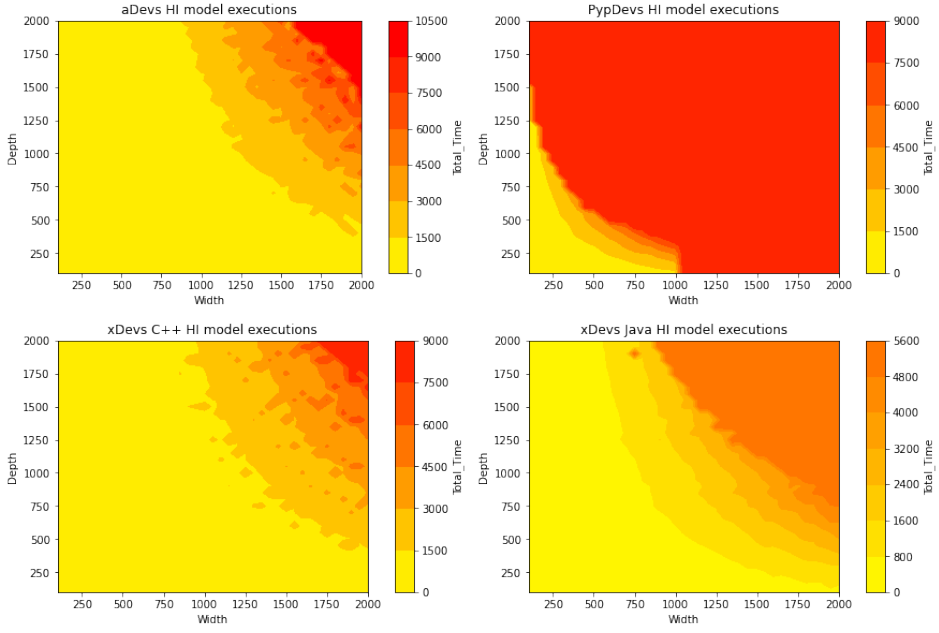


Figura 4.2: Mapas de calor para las ejecuciones del modelo HI.

Analizando los mapas de calor anteriores podemos llegar a varias conclusiones importantes. A la vista de los resultados obtenidos para el modelo LI (figura 4.1), es claro que tanto aDevs como xDevs, en sus implementaciones en C++ o Java, presentan mejor desempeño en las ejecuciones de las simulaciones frente a PypDevs. De hecho, tal y como se comentó ya en la anterior subsección, en PypDevs la mayoría de instancias fallan debido a la incapacidad del simulador para lidiar con tamaños grandes de anchura o profundidad. La comparativa entre xDevs y aDevs se complica por la escala que impone el modelo PypDevs, que claramente requiere de un mayor tiempo de ejecución incluso en instancias pequeñas de simulaciones. Por ello, procedemos a incluir la figura 4.5 que muestra los resultados escalados sin considerar las ejecuciones de PypDevs.

A partir de lo expuesto en la figura 4.5 se puede observar que el simulador que, a rasgos generales, se comporta mejor en las ejecuciones del modelo LI es aDevs, seguido de xDevs en su versión en C++ con un desempeño

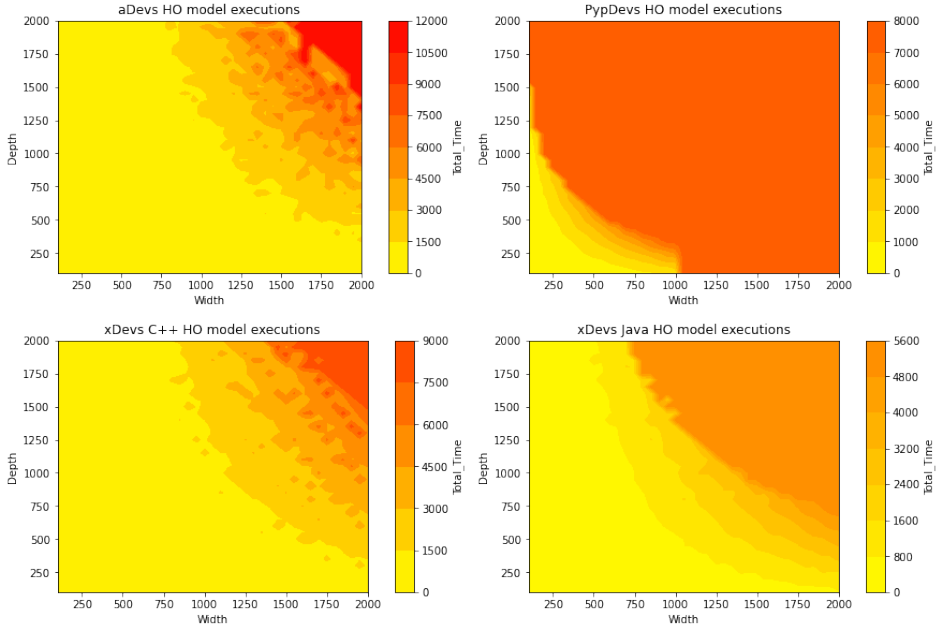


Figura 4.3: Mapas de calor para las ejecuciones del modelo HO.

que duplica el de su implementación en Java para multitud de instancias.

Fijándonos ahora en la figura 4.2, observamos claramente que PypDevs arroja un peor desempeño que el resto de simuladores, a la par que presenta multitud de instancias fallidas para la mayoría de tamaños de profundidad y anchura propuestos. Sin embargo, la diferencia no es tan drástica como ocurría con LI, siendo en esta situación la escala adecuada para comparar todos los simuladores juntos. De forma general, podemos decir que xDevs en su implementación en Java arroja mejores resultados, sobre todo en instancias de gran anchura y profundidad, frente a la implementación en C++ o al simulador aDevs. A su vez, el desempeño de xDevs C++ se presenta ligeramente mejor que aDevs, aunque el comportamiento en las simulaciones es semejante.

Pasemos ahora a analizar las simulaciones del modelo HO, representadas en el mapa de calor de la figura 4.3. En este caso, al igual que en HI, la escala es adecuada para todas las ejecuciones, pudiéndose comparar

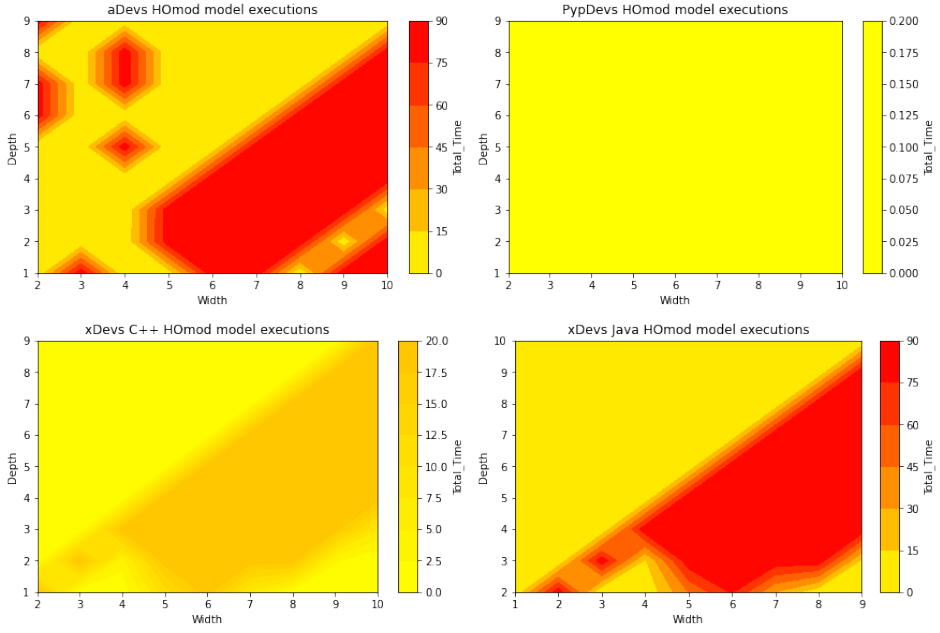


Figura 4.4: Mapas de calor para las ejecuciones del modelo HOmod.

claramente los simuladores. En este caso xDevs Java parece comportarse bastante bien. Además, notemos que esta vez PypDevs no presentó ejecuciones fallidas, mientras que aDevs sí lo hizo en las instancias de mayor anchura y profundidad. De hecho, el desempeño de PypDevs parece incrementarse notablemente con respecto a lo observado en los modelos LI y HI. Notemos que HO se define como un modelo con un alto nivel de acoplamiento, al igual que HI, pero que a su vez incorpora un mayor número de salidas. A partir de los resultados analizados hasta ahora parece claro que PypDevs se comporta mejor en modelos de mayor complejidad, con alto grado de acoplamiento y numerosas salidas. Por contra, en modelos de pocas interconexiones o moderadamente alto grado de acoplamiento en las entradas del modelo el desempeño es pobre.

La evolución contraria parece darse en los simuladores aDevs y xDevs, que para modelos de complejidad inferior arrojaron mejores tiempos de ejecución. Este deterioro se ve claramente más acentuado en el caso del

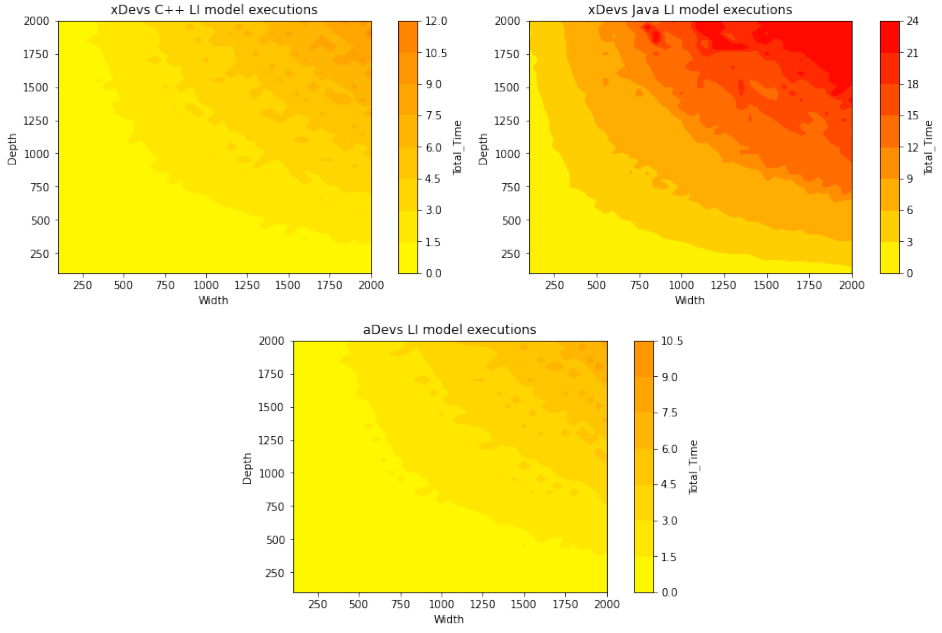


Figura 4.5: Mapas de calor para las ejecuciones del modelo LI, sin PypDevs.

simulador aDevs, seguido de xDevs en su implementación en C++.

Finalmente, pasemos a estudiar las simulaciones del modelo HOmod. Recordemos que este modelo presenta un nivel exponencial de acoplamiento, siendo las instancias simuladas de menor profundidad y anchura para evitar problemas de memoria debido a su comportamiento complejo que además requiere una intensa potencia de cómputo. Los mapas de calor de la figura 4.4 muestran los resultados obtenidos en las simulaciones. En este caso ocurre lo contrario que pudimos apreciar para el modelo LI, la escala no permite analizar los simuladores por presentar PypDevs un mejor desempeño general en las simulaciones. Sin embargo, a pesar de que la escala no permite visualizar el comportamiento de PypDevs claramente, sí que nos permite comparar los modelos entre sí. Así, podemos observar que, objetivamente, PypDevs ofrece mejores tiempos de ejecución total en todas las simulaciones realizadas frente a los simuladores xDevs y aDevs. De hecho, tanto aDevs como xDevs en su implementación en Java presentaron

errores en instancias de gran profundidad.

Se puede apreciar además una línea divisoria entre ejecuciones que se ejecutan generalmente rápido y aquellas que más cuestan o que directamente produjeron errores. Esta línea divisoria que puede observarse en los mapas de calor indica que la complejidad de simular HOmod aumenta drásticamente con respecto a su anchura en mayor medida que frente al aumento en profundidad. Este hecho se correlaciona directamente con la complejidad computada en la subsección 2.2.5 donde vimos que HOmod depende en un orden de magnitud mayor del ancho que de la profundidad.

A modo de complemento de las observaciones anteriores, con respecto a HOmod, incluimos el mapa de calor de la ejecución de PypDevs de HOmod sin escalar los colores para permitir la comparación con otros simuladores. La figura 4.6 muestra este mapa de calor.

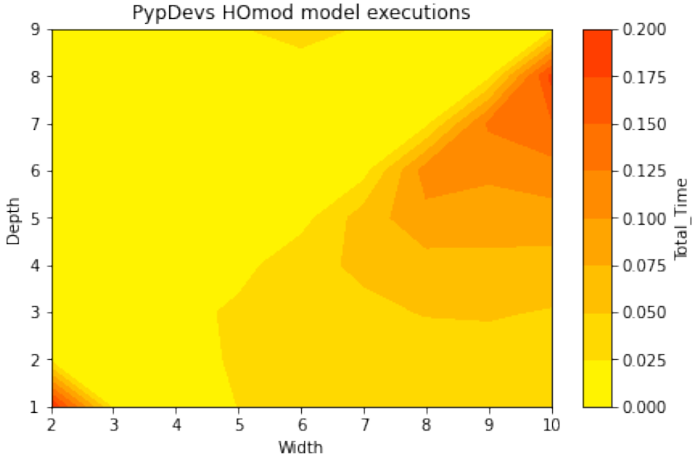


Figura 4.6: Mapa de calor de HOmod para PypDevs.

Al igual que ocurría con xDevs y aDevs, puede apreciarse con la nueva escala que en PypDevs también se encuentra una línea que divide los tiempos de ejecución del modelo en función de la anchura y profundidad. Así, claramente para mayores anchuras el modelo requiere un cómputo más intensivo que frente a instancias de gran profundidad y menor anchura. Esto es, tal y como vimos, en general el factor limitante para las simulaciones de HOmod es principalmente la profundidad.

A modo de resumen, podemos observar que, a partir de los mapas de calor expuestos, se pueden derivar ciertas conclusiones con respecto a los simuladores. Entre ellas destacamos que es claro el buen comportamiento de PypDevs en las ejecuciones de modelos complejos como HOmod o HO, donde destaca de forma muy notable con respecto a los otros simuladores. Por el contrario, en modelos simples como LI o HI, PypDevs presenta un desempeño bastante pobre, siendo aDevs y xDevs claramente superiores. También, podemos observar que tanto aDevs como xDevs empeoran los tiempos totales en simulaciones complejas de forma drástica con respecto a las simulaciones de modelos simples. De hecho, aDevs ejecuta mejor que cualquier otro simulador instancias de LI, pero es a su vez el peor para instancias de HO o HOmod.

4.2. Métricas y estimadores

Para complementar el análisis de la sección 4.1, pasaremos ahora a estudiar las métricas y estimadores propuestos para el estudio del rendimiento de los simuladores. Para ello, haremos uso de las métricas expuestas en 3.2.2 y 3.2.3.

4.2.1. Métricas sobre los modelos

Debido a la cantidad de fallos en las simulaciones ejecutadas expuestas en la subsección 4.1.1, consideraremos dos alternativas para el cómputo de las métricas $\overline{t^{m,s}}$ para cada modelo m y simulador s. Por un lado, evaluaremos solamente las instancias ejecutadas, ignorando los errores y obtendremos un valor $\overline{t_1^{m,s}}$. Por el otro lado, calcularemos las instancias ejecutadas asignando a las simulaciones fallidas el tiempo máximo de ejecución de ese modelo por alguno de los simuladores, obteniendo otra métrica $\overline{t_2^{m,s}}$. Notemos que en este caso $\overline{t_2^{m,s}}$ es dependiente de la prueba analizada, al ser los fallos en las simulaciones consecuencia del *hardware* utilizado en el test de rendimiento planteado. Sin embargo, esta comparativa puede verse más justa cuando los dispositivos físicos limitan los resultados disponibles. En un escenario ideal, habrían de ejecutarse las simulaciones correctamente para todos los simuladores y tamaños de modelos escogidos. Los resultados de ambas métricas se recogen en las tablas 4.4, 4.5, 4.6 y 4.7.

aDevs

Modelo	$\overline{t_1^{m,aDevs}}$
LI	0.999988006295817
HI	0.9999999892713667
HO	0.9999999892574057
HOmod	0.9940390480081167

PypDevs

Modelo	$\overline{t_1^{m,PypDevs}}$
LI	0.9999711936970835
HI	0.9999998423510298
HO	0.9999998380650436
HOmod	0.9991200837185923

Cuadro 4.4: Métrica $\overline{t_1^{m,s}}$ de modelos para aDevs y PypDevs.

xDevs-C++

Modelo	$\overline{t_1^{m,xDevs-C++}}$
LI	0.9999882112195594
HI	0.9999999899200893
HO	0.9999999892188497
HOmod	0.996002323836742

xDevs-Java

Modelo	$\overline{t_1^{m,xDevs-Java}}$
LI	0.9999881955762842
HI	0.9999999816472106
HO	0.9999999771843888
HOmod	0.9967858091333365

Cuadro 4.5: Métrica $\overline{t_1^{m,s}}$ de modelos para xDevs C++ y Java.

aDevs

Modelo	$\overline{t_2^{m,aDevs}}$
LI	0.999988006295817
HI	0.9999999905968371
HO	0.9999999907011534
HOmod	0.9992971998721605

PypDevs

Modelo	$\overline{t_2^{m,PypDevs}}$
LI	0.9999954101973322
HI	0.9999999923565613
HO	0.999999992480811
HOmod	0.9991200837185923

Cuadro 4.6: Métrica $\overline{t_2^{m,s}}$ de modelos para aDevs y PypDevs.

La comparación visual entre los valores recogidos en las tablas anteriores no es sencillo. En su lugar, estudiaremos una comparativa utilizando gráficos de barras. La figura 4.7 representa los datos utilizando dicho es-

xDevs-C++		xDevs-Java	
Modelo	$\overline{t_2^{m,xDevs-C++}}$	Modelo	$\overline{t_2^{m,xDevs-Java}}$
LI	0.9999882112195594	LI	0.9999897290448272
HI	0.9999999906249357	HI	0.9999999910344126
HO	0.9999999907844583	HO	0.9999999912043782
HOmod	0.9993073085517644	HOmod	0.9994095900471597

Cuadro 4.7: Métrica $\overline{t_2^{m,s}}$ de modelos para xDevs C++ y Java.

quema. Notemos que claramente los valores de la métrica para HOmod son sustancialmente menores que los del resto de modelos, impidiendo una comparativa entre los demás. Por ello, incluimos también la figura 4.8 que desglosa los valores de la métrica escalando para cada modelo los datos, y permitiendo así una visualización de los datos adecuada.

Nótese que, por culpa de los errores, los valores de las métricas quedan distorsionados. Así, como en la mayoría de instancias grandes PypDevs falló para el modelo LI, en los gráficos anteriores se ve beneficiado al haberse podido ejecutar solo para instancias pequeñas con mejores tiempos que el resto. De esta manera, el estimador $\overline{t_1^{m,s}}$ sufre del problema que auguramos con respecto a los valores de las simulaciones fallidas. Otro ejemplo claro son las simulaciones de HOmod, que para aDevs y xDevs fallaban muchas, mientras que para PypDevs no. Así, parece que PypDevs presenta un peor desempeño por haber podido ejecutar más instancias de mayor complejidad, que en vista de lo expuesto en 4.1.2 tomaron además más tiempo que las instancias pequeñas. Para estudiar mejor entonces la relación entre simuladores conviene utilizar la segunda métrica propuesta $\overline{t_2^{m,s}}$, que maneja también las instancias fallidas, penalizándolas.

De forma análoga a como expusimos la métrica $\overline{t_1^{m,s}}$, incluimos a continuación las figuras 4.9 y 4.10 que muestran la comparativa general de la métrica para todos los simuladores y modelos, y un desglose para escalar los datos ajustando a los resultados por modelo.

Notemos ahora que para estas figuras el valor de la métrica parece correlacionarse de forma coherente con las conclusiones obtenidas de los mapas de calor analizados. A saber, podemos observar que claramente PypDevs

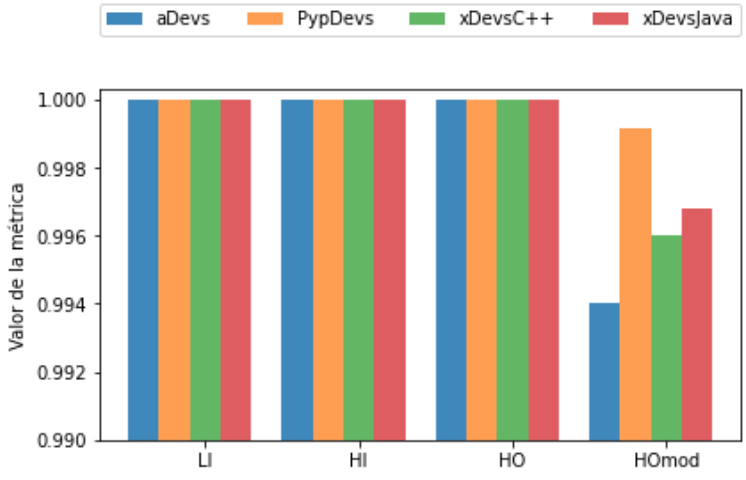


Figura 4.7: Valores de $\overline{t_1^{m,s}}$ para los distintos simuladores y modelos.

tiene mucho mejor desempeño, con un valor del estimador mucho menor, que aDevs o xDevs para HOmod. A su vez, se ve que este mismo simulador ejecuta peor instancias de modelos simples como LI o HI. También, se puede apreciar claramente que xDevs Java ejecuta mejor las instancias simuladas de HI y HO pero no de HOmod frente a C++, tal y como ya se adelantaba en las subsecciones anteriores.

Es importante observar que a partir de los resultados analizados podemos concluir que la comparación empleando la métrica propuesta no es adecuada cuando hay simulaciones fallidas presentes. En estos casos, que generalmente suelen ser en las instancias límites de anchura o profundidad, se sobreestima o subestima el rendimiento de un simulador con respecto a un modelo. Para estos casos, un enfoque como el adoptado en esta subsección podría servir de apoyo para resolver el problema y permitir una comparación justa. Sin embargo, tal y como ya mencionamos, es importante destacar que los resultados arrojados serán dependientes del tiempo de ejecución máximo alcanzado entre todos los simuladores antes del fallo, para las instancias que terminaron incorrectamente. Así, los resultados numéricos de la métrica para estos simuladores no son comparables directamente en estudios futuros.

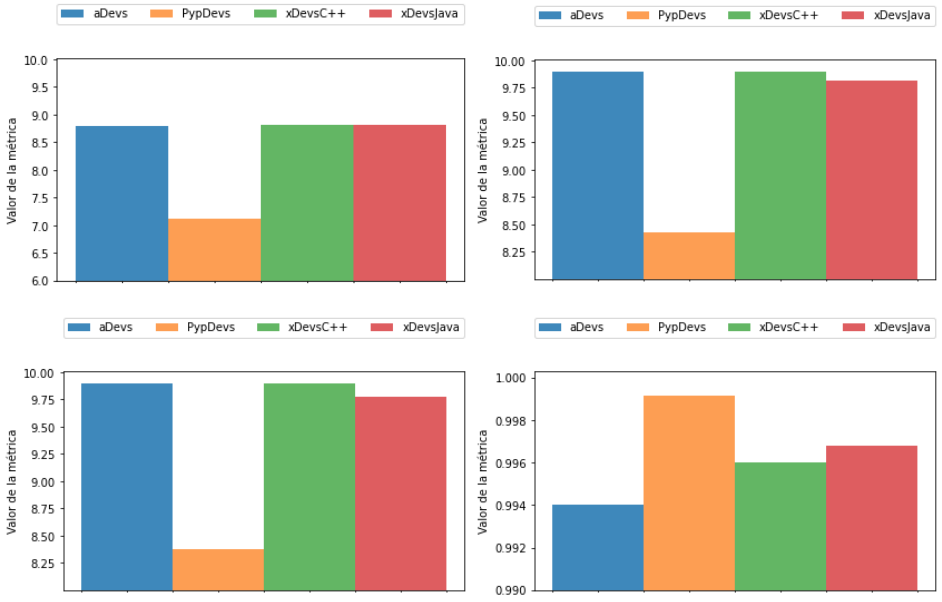


Figura 4.8: Métrica $\overline{t_1^{m,s}}$ para los simuladores separados para ajustarse a cada escala.

Finalmente, podemos destacar que a partir de los resultados obtenidos es posible concluir que la métrica empleada refleja de forma fiel los resultados cualitativos que podían derivarse de los mapas de calor analizados. Así, el comportamiento de PypDevs es mejor en modelos complejos frente a simples. De forma análoga, los comportamientos de aDevs y xDevs son parecidos. Además, por construcción propia de la métrica empleada, es posible comparar los diferentes valores arrojados utilizando razones.

4.2.2. Métricas de rendimiento global

Notemos que los resultados obtenidos en la subsección anterior son referidos a cada modelo y al comportamiento de cada simulador para instancias concretas. Así, cuestiones como la idoneidad de un simulador para un cierto tipo de problema o la ejecución de un modelo con un tamaño concreto, pueden responderse de forma adecuada a través del estudio de las métricas

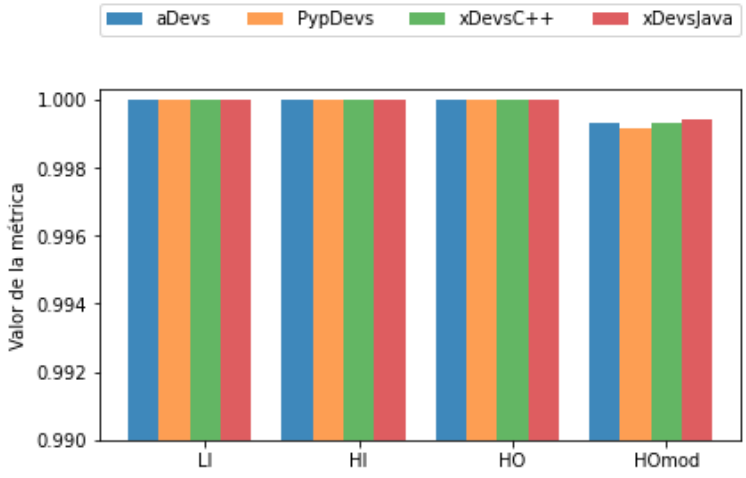


Figura 4.9: Valores de $\overline{t_2^{m,s}}$ para los distintos simuladores y modelos.

anteriores. Sin embargo, otras cuestiones como cuál es el mejor simulador en general no pueden responderse con métricas locales, sino que requieren de una agregación global de los valores obtenidos para las simulaciones de los distintos modelos.

Recordemos que la fórmula de agregación propuesta para la estimación global se recoge en la ecuación (3.2) y que está derivada de la métrica geométrica anterior. La figura 4.11 muestra una gráfica comparativa de la métrica global computada para cada simulador.

A partir de los valores computados de $\overline{r_s}$ es posible, además, obtener el valor de rendimiento global r_s normalizando con respecto de una referencia. En nuestro caso, escogeremos aDevs como simulador de referencia. La figura 4.12 muestra una comparación del rendimiento global tras normalizar con respecto de aDevs. La gráfica ilustra los valores de la métrica r_s para los distintos simuladores. En este caso, aDevs actúa de escala, siendo los simuladores que en el gráfico se encuentran por encima de su barra mejores, y en caso contrario peores en rendimiento. Esto ocurre ya que al normalizar con respecto a un simulador, los valores superiores a 1 indican que el simulador s presenta, en general, un mejor rendimiento que con aDevs. Por otro lado, valores inferiores a 1 muestran que el comportamiento del

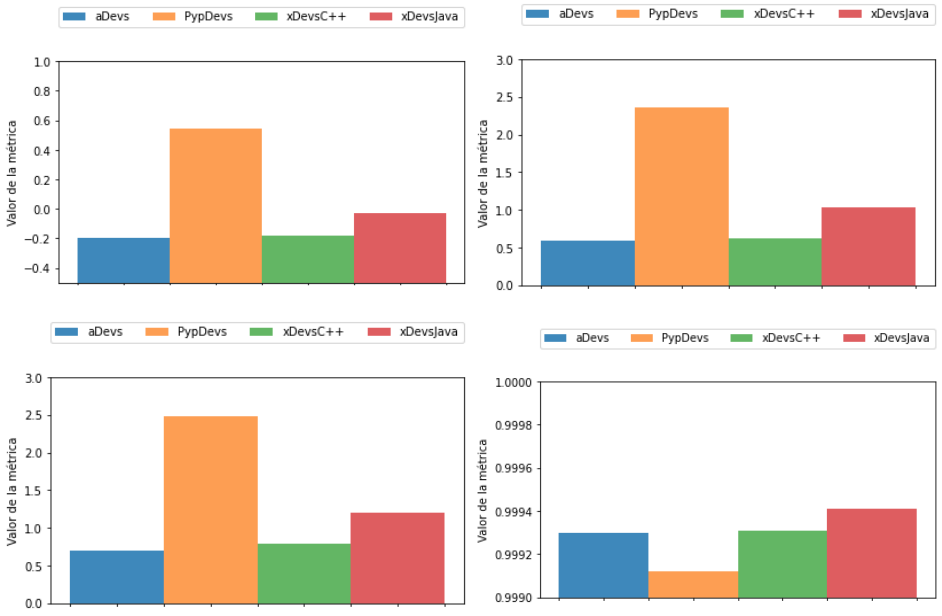


Figura 4.10: Métrica $\overline{t_2^{m,s}}$ para los simuladores separados para ajustarse a cada escala.

simulador es peor que con aDevs.

Podemos concluir a partir de la figura anterior que el mejor simulador, en general, es PypDevs, seguido por aDevs y posteriormente por xDevs en su implementación en C++. El peor simulador de forma general es xDevs en su versión en Java. Este hecho se correlaciona con el análisis realizado hasta ahora. Como hemos visto, xDevs en C++ y aDevs presentan un comportamiento similar, mientras que PypDevs destaca excepcionalmente en modelos complejos. Al agregar los datos, PypDevs se ve beneficiado por ser excepcionalmente bueno en modelos complejos y solamente excepcionalmente malo en la ejecución de LI. Por otro lado, xDevs Java era excepcionalmente malo en las ejecuciones de HOmod y también ofrecía resultados pobres en comparación con aDevs y xDevs C++ para LI. Esto lo ha penalizado con respecto al resto de simuladores, arrojando un valor de rendimiento inferior. De hecho, esto pone de manifiesto un comportamiento

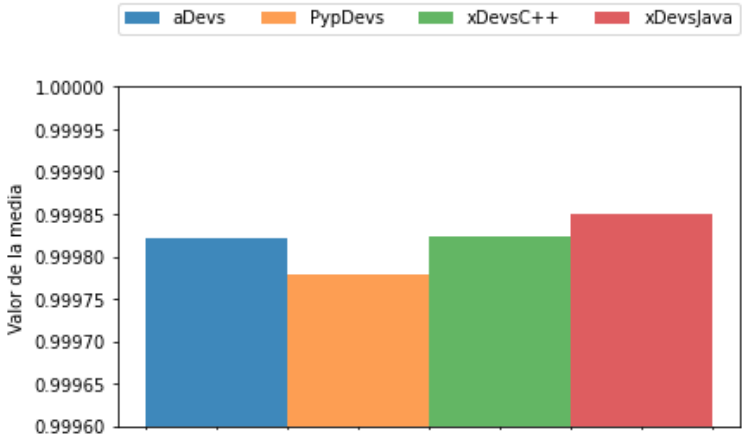


Figura 4.11: Valores de $\overline{r_s}$ para los distintos simuladores.

relativamente peor con este simulador tanto en las instancias más simples, con LI, como en las más complejas, con HOmod.

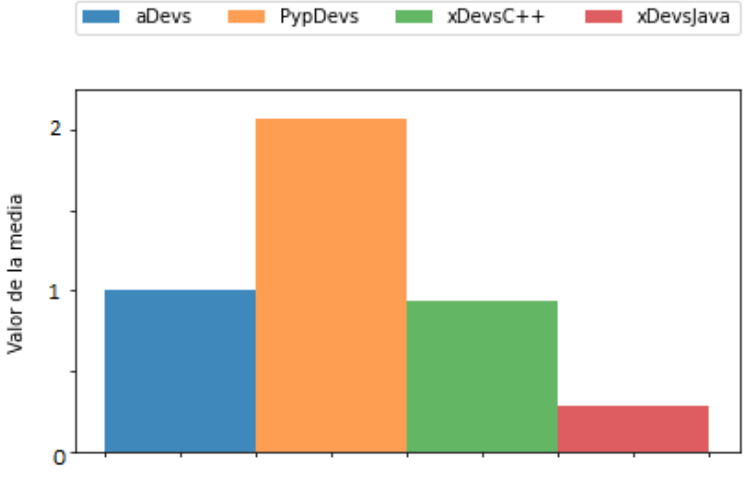


Figura 4.12: Valores de r_s para los distintos simuladores.

Capítulo 5

Conclusiones

En este capítulo se recoge la revisión de los objetivos planteados en la sección 1.2, así como un resumen general de las principales conclusiones que se derivan de este estudio a modo de consideraciones finales. También, se incluyen propuestas de investigaciones para continuar con este trabajo.

5.1. Revisión de los objetivos

Durante todo el documento se han ido cubriendo los diferentes objetivos fijados al inicio del trabajo. En esta sección relacionaremos de forma explícita los objetivos generales y específicos planteados con los capítulos, secciones y subsecciones redactadas. De esta manera, revisaremos cómo se han ido alcanzando todas y cada una de las metas que se marcaron.

Estudio del formalismo DEVS y simuladores disponibles

El objetivo general [O1] marcaba como meta el estudio de DEVS y de las principales implementaciones disponibles. Precisamente, en el capítulo 2 de estudio del estado del arte, se cubrió este objetivo general. De forma pormenorizada, podemos observar que el objetivo [O1.1] se alcanzaba en la subsección 2.1.1, mientras que los objetivos [O1.2] y [O1.3] se completaban en las subsecciones 2.1.2 y 2.1.3.

Investigar el análisis de rendimiento de sistemas

En referencia con el objetivo [O2], destacamos que se buscaba poder comprender cómo se había abordado la cuestión del rendimiento en la literatura, examinando las diferentes técnicas utilizadas en el campo y las métricas existentes. Este objetivo se lograba alcanzar a través del estudio expuesto en la sección 2.2. En particular, el objetivo específico [O2.1] se lograba al comienzo de dicha sección, mientras que [O2.2] se detallaba en las subsecciones 2.2.1, 2.2.2 y 2.2.3.

Descripción y estudio del *benchmark* DEVStone

El objetivo general [O3] buscaba poder destacar el principal *benchmark* ya existente para el análisis del rendimiento en modelos DEVS. Este objetivo y sus correspondientes subobjetivos más específicos [O3.1] y [O3.2] fueron completados a través de las subsecciones 2.2.4 y 2.2.5.

Propuesta de métrica para la evaluación del rendimiento

Con objeto de poder satisfacer la tarea inicial que motivaba este trabajo, se buscó proponer una métrica adecuada para el análisis del rendimiento en simuladores de eventos discretos. Este objetivo se recogió en [O4], a través de diferentes objetivos específicos. Estas metas fueron alcanzadas a través de las secciones 2.2 y 3.2. En concreto, el objetivo específico [O4.1] se satisfacía con la sección 2.2 y 3.2.2, mientras que [O4.2] y [O4.3] se completaron con los resultados expuestos en el capítulo 4.

Comparativa del rendimiento entre simuladores

El objetivo general [O5] destacaba la importancia de poder aplicar la obtención de una métrica adecuada al análisis y comparativa de simuladores basados en DEVS. Para ello, se buscó fijar de forma explícita el poder realizar una comparativa real de los principales simuladores e implementaciones DEVS como meta. Esta tarea se completaba a través de lo expuesto en los capítulos 3 y 4. De forma pormenorizada, podemos ver que el objetivo [O5.1] se cubría a través de lo expuesto en la sección 3.1, mientras que [O5.2] y [O5.3] se alcanzaron en 4.1 y 4.2.

Alcance, limitaciones y futuras vías de investigación

Este último objetivo general [O6] se puede ver detallado a través de las consideraciones realizadas en los capítulos 3 y 4 con respecto al análisis de los resultados y la metodología empleada. Además, en la sección que concluye este capítulo (sección 5.2), se resume el alcance, las limitaciones, y la viabilidad de este estudio. También, se incluye en dicha sección una propuesta de trabajo futuro para continuar con la investigación iniciada y recogida en este documento.

5.2. Consideraciones finales y trabajo futuro

Examinando los diferentes capítulos que cubren este documento, podemos determinar que se han conseguido completar con éxito todos los objetivos propuestos. Es más, la cuestión que surgía como motivación inicial ha quedado saciada a través del planteamiento y el estudio de las diferentes métricas empleadas en el análisis del rendimiento de los simuladores. Pasaremos ahora a resumir las principales conclusiones que se derivan de todo el trabajo, y posteriormente comentaremos posibles vías para continuar investigando en la dirección marcada hasta ahora.

5.2.1. Resumen de resultados

A partir de lo analizado en el capítulo 4, podemos concluir que la propuesta de métrica realizada es adecuada para la evaluación del rendimiento de simuladores de eventos discretos. Esta métrica es idónea dadas sus propiedades adquiridas por propia construcción, haciéndola adecuada para la tarea objetivo. Complementando al análisis teórico, hemos visto que ha sido también adecuada en nuestra evaluación particular de los simuladores aDevs, xDevs C++, xDevs Java y PypDevs.

Observando la subsección 3.2.1 vimos que la mayoría de ejecuciones de modelos simples fallaban para PypDevs, mientras que las ejecuciones de modelos complejos daban error para el resto de simuladores. Asimismo, podemos destacar también que el comportamiento de aDevs y xDevs C++ era en general muy similar en cuanto a número y tipo de errores de ejecución en las simulaciones. En contraposición, xDevs Java presentaba peores

resultados que ambos.

Tras esto, pasamos a analizar en la subsección 4.1.2 los mapas de calor de las simulaciones ejecutadas. Visualizando los datos, pudimos concluir claramente que PypDevs ejecutaba mejor instancias de modelos complejos, y de forma excepcionalmente mala los modelos simples. Igualmente, vimos que el comportamiento de aDevs y xDevs C++ era similar, frente a xDevs Java que presentaba en líneas generales un peor desempeño. Estos hechos corroboraron las conjeturas realizadas en la subsección anterior mientras analizamos los fallos totales.

Finalmente, en la sección 4.2 pasamos a evaluar las métricas propuestas sobre los simuladores y los modelos. Confirmamos entonces la situación descrita anteriormente a través de una estimación numérica, pudiendo afirmar que efectivamente PypDevs ofrecía un mejor desempeño en modelos complejos frente a aDevs y xDevs. Además, pudimos observar cuantitativamente la relación de similitud entre aDevs y xDevs C++, así como la disimilitud existente con xDevs Java. Para concluir, agregando de forma global los resultados arrojados para todos los modelos de cada simulador, pudimos observar que el mejor simulador en términos generales fue PypDevs. Esto se debe a que aun cuando las instancias pequeñas las ejecutó excepcionalmente mal, en el caso de las instancias medianamente o muy complejas su desempeño fue bueno o excepcionalmente bueno. También, corroboramos la similitud entre xDevs C++ y aDevs, así como la diferencia con xDevs Java que se comportaba notablemente peor con instancias muy simples o muy complejas.

5.2.2. Trabajo futuro

A pesar de las conclusiones anteriormente expuestas, es destacable que los resultados y conclusiones derivadas de este trabajo constituyen únicamente un hito más en el camino hacia la elaboración de *benchmarks* y hojas de especificaciones para el rendimiento de los simuladores de eventos discretos. Sin embargo, este hito requiere de un estudio continuado para seguir desarrollando el campo en la dirección marcada hasta ahora. Así, este trabajo constituye una base sobre la que seguir avanzando en la evaluación del rendimiento de simuladores.

Al igual que ocurre con la organización SPEC [8], como continuación de

este trabajo, podría elaborarse una *suite* completa de *benchmarks* y métricas que se constituyesen como estándar en la academia e industria para la evaluación de simuladores de eventos discretos. Para conseguir esto, se pueden emplear muchas de las conclusiones obtenidas en este trabajo, así como se puede recurrir a las métricas y técnicas propuestas para la evaluación del rendimiento. Los resultados obtenidos podrían entonces combinarse para definir formalmente una hoja de evaluación del rendimiento de simuladores, de manera que cualquier usuario pueda estimar el desempeño de un determinado sistema. Es más, la comparación podría darse también entre diferentes versiones de un mismo simulador.

Por otro lado, aunque en este documento se ha mencionado también las limitaciones de algunos simuladores con respecto al uso y gestión de la memoria, sería interesante construir tests y pruebas que evalúen en función de este criterio. Así, podría complementarse la comparativa en cuanto a rendimiento estimando el desempeño en función del uso y gestión de la memoria o el coste de ejecución de la simulación. Esta estimación podría darse a través de la evaluación de métricas que midan las ejecuciones fallidas en función del tamaño de los modelos, el coste en memoria de ejecución de cada instancia o el uso de la CPU más o menos intensivo empleado en cada simulación. Utilizando diferentes aproximaciones como estas puede obtenerse una imagen más realista de los simuladores, permitiendo una comparación más justa al contemplar desde distintas ópticas el funcionamiento real de los simuladores.

Finalmente, podemos destacar que el estudio comparativo presente se encuentra limitado por los simuladores empleados, y a versiones concretas. Así, podría ser interesante comparar diferentes versiones de un mismo simulador para estudiar su evolución, y si existe una mejora real en el rendimiento en las diferentes actualizaciones. También, puede replicarse el análisis realizado para otras librerías y simuladores como CD++ [45], GALATEA [16], PowerDEVS [13], DEVSim++ [29], SimStudio [38] o MS4Me [36], entre otros. Notemos que el número de herramientas, simuladores y librerías disponibles que utilicen DEVS está en continuo aumento, a la par que se actualizan sus diferentes versiones. Por este motivo, existe una necesidad constante de comparación y evaluación.

Capítulo 6

Conclusions

After reviewing the different chapters of this document, we can firmly say that we met all of the objectives initially proposed. Moreover, the problem of determining a metric for the performance analysis of simulators has been successfully addressed. We will now move to sum up the main conclusions derived from the work carried out. After that, we will expose the different perspectives that can be considered to continue this work.

6.1. Final remarks

Taking into consideration what we described in Chapter 4, we can conclude that the metric proposed is suitable for the performance evaluation of discrete event simulators. The metric is ideal for the task at hand due to its natural properties. Following up the theoretical analysis of the metric, we also apply it to a concrete study case. Therefore we were able to evaluate aDevs, xDevs C++, xDevs Java y PypDevs.

If we take into account what we saw in Subsection 3.2.1, we can say that the majority of PypDevs execution of simple models failed while most complex models are successfully run. In addition, we highlight that the behavior of aDevs and xDevs C++ was similar in terms of number and type of execution errors. In contrast, xDevs Java offered a worse performance when compared with the previous two.

Following up on the previous analysis, we move to Subsection 4.1.2 whe-

re we presented the heatmaps of the results. Visualizing the data, we could conclude that PypDevs was clearly better when running complex instances, while it was the worts at simple ones. At the same time, we concluded that aDevs and xDevs C++ were really similar, in contrast to xDevs Java which offered a worse performance in general terms. These considerations corroborated the conjectures made in the previous subsection.

Finally, in Section 4.2 we applied the metrics proposed to evaluate the simulators when running the different models. The results supported the previous considerations, by allowing us to numerically describe the relationship among the simulators. In this sense, we saw that PypDevs offered a better performance when running complex models. Also, aDevs and xDevs C++ were numerically similar in their performances. Moreover, xDevs Java was as well numerically different from these two. To conclude, we aggregated all the results obtained from the different models, and we could conclude which simulator was better. In general terms, the best one was PypDevs. This might be due to the great performance of this simulator for huge and complex instances. In addition, we corroborated the similarity among xDevs C++ and aDevs, which behaved comparably, as well as the dissimilarity with xDevs Java (which performed notably wrong in small instances or complex ones).

Agradecimientos

El trabajo realizado en este proyecto ha venido respaldado por el programa nº GCP19980904 de créditos de investigación de Google Cloud.

Acknowledgments

This material is based upon work supported by the Google Cloud Research Credits program with the award GCP19980904.

Bibliografía

[1] *aDEVS: A discrete event system simulator*. https://web.ornl.gov/~nutarojj/adevs/.

[2] *aDEVS. A simulation library for discrete event and hybrid dynamic systems*. https://sourceforge.net/p/adevs/code/HEAD/tree/.

[3] *Debian*. https://www.debian.org/.

[4] *Google Colaboratory*. https://colab.research.google.com/.

[5] *proc(5) — Linux manual page*. https://man7.org/linux/man-pages/man5/proc.5.html.

[6] *Slurm workload Manager*. https://slurm.schedmd.com/.

[7] *SPECviewperf® 7.1 – Weighted Geometric Mean*. https://www.spec.org/gwpg/gpc.static/geometric.html.

[8] *Standard performance Evaluation Corporation*. https://www.spec.org/.

[9] *The CentOS project*. https://www.centos.org/.

[10] *Tmux(1) - Linux manual page*. https://man7.org/linux/man-pages/man1/tmux.1.html.

[11] Amdahl, Gene M.: *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. AFIPS '67 (Spring), página 483–485, New York, NY, USA, 1967. Association for Computing Machinery, ISBN 9781450378956. https://doi.org/10.1145/1465482.1465560.

[12] Babulak, Eduard y Ming Wang: *Discrete Event Simulation: State of the Art*. En *Discrete Event Simulations*. Sciyo, Agosto 2010. https://doi.org/10.5772/9894.

[13] Bergero, Federico y Ernesto Kofman: *PowerDEVS: A tool for hybrid system modeling and real-time simulation*. Simulation, 87:113–132, Enero 2011.

[14] Byon, Eunshin, Eduardo Pérez, Yu Ding y Lewis Ntaimo: *Simulation of Wind Farm Operations and Maintenance using Discrete Event System Specification.* Simulation, 87:1093–1117, Diciembre 2011.

[15] Cárdenas, Román, Patricia Arroba y José L. Risco-Martín: *Bringing AI to the edge: A formal MS specification to deploy effective IoT architectures*. Journal of Simulation, 2021.

[16] Dávila, Jacinto A. y Mayerlin Uzcágegui: *GALATEA: A multi-agent, simulation platform*. 2000.

[17] Dobniewski, Alejandro y Gastón Christen: *Representación gráfica de modelos DEVS y modificaciones a CD++ para su simulación*. 2003.

[18] Faradj, Mabrouck K.: *Which mean do you mean?: an exposition on means*, 2004. https://digitalcommons.lsu.edu/gradschool_theses/1852.

[19] Fleming, Philip J. y John J. Wallace: *How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results*. Commun. ACM, 29(3):218–221, Marzo 1986, ISSN 0001-0782. https://doi.org/10.1145/5666.5673.

[20] Gardner, Martin: *Mathematical Games*. Scientific American, 223(4):120–123, Octubre 1970. https://doi.org/10.1038/scientificamerican1070-120.

[21] Glinsky, E. y G. Wainer: *DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments*. En *Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications*, páginas 265–272, 2005.

[22] Harris, Charles R., K. Jarrod Millman *y cols.*: *Array programming with Numpy*. Nature, 585(7825):357–362, Septiembre 2020. https://doi.org/10.1038/s41586-020-2649-2.

[23] Henares, Kevin, José L. Risco-Martín, Román Hermida, Gemma Reig Roselló y Román Cárdenas: *Modular framework to model critical events in stroke patients*. En *Proceedings of the 2019 Summer Simulation Conference (SummerSim 2019)*, 2019.

[24] Hunter, J. D.: *Matplotlib: A 2D graphics environment*. Computing in Science & Engineering, 9(3):90–95, 2007.

[25] Iscar-Ucm: *XDEVS: A cross-platform discrete event system simulator*. https://github.com/iscar-ucm/xdevs.

[26] Jacob, Bruce y Trevor Mudge: *Notes on Calculating Computer Performance*. Informe técnico, 1995.

[27] John, Lizy Kurian: *More on Finding a Single Number to Indicate Overall Performance of a Benchmark Suite*. ACM Computer Architecture News, 2004.

[28] Jones, Eric, Travis Oliphant, Pearu Peterson *y cols.*: *SciPy: Open source scientific tools for Python*, 2001–. http://www.scipy.org/.

[29] Kim, Tag, Chang Sung, Su Youn Hong, Jeong Hee Hong, Changbeom Choi, Jeong Kim, Kyung Min Seo y Jang Won Bae: *DEVSim++ Toolset for Defense Modeling and Simulation and Interoperation*. The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology, 8:129–142, Julio 2011.

[30] Kistowski, Jóakim von, Jeremy Arnold, Karl Huppler, Klaus Dieter Lange, John Henning y Paul Cao: *How to Build a Benchmark*. Febrero 2015.

[31] Krejčí, Jana y Jan Stoklasa: *Aggregation in the analytic hierarchy process: Why weighted geometric mean should be used instead of weighted arithmetic mean*. Expert Systems with Applications, 114:97–106, 2018, ISSN 0957-4174. https://www.sciencedirect.com/science/article/pii/S0957417418303981.

[32] McKinney, Wes: *Data Structures for Statistical Computing in Python* . En Walt, Stefan van der y Jarrod Millman (editores): *Proceedings of the 9th Python in Science Conference* , páginas 56 – 61, 2010.

[33] Molero, Xavier, Carlos Juiz y Miguel Jesús Rodeño: *Evaluación y modelado del rendimiento de los sistemas informáticos*. 2004.

[34] Pérez-Vilarelle, Laura, José L. Risco-Martín y José L. Ayala: *Modeling and simulation of wind energy production in the smart-grid scenario*. En *Proceedings of the 2018 Spring Simulation Multionference (SpringSim 2018)*, páginas 2:1–2:12, 2018.

[35] Risco-Martín, José L., Saurabh Mittal, Juan Carlos Fabero Jiménez, Marina Zapater y Román Hermida Correa: *Reconsidering Performance of DEVS Modeling and Simulation Environments Using the DEVStone Benchmark*. 2017.

[36] Seo, Chungman, Bernard Zeigler, Robert Coop y Doohwan Kim: *DEVS modeling and simulation methodology with MS4 Me software tool*. Volumen 45, Abril 2013.

[37] Smith, J. E.: *Characterizing Computer Performance with a Single Number*. Commun. ACM, 31(10):1202–1206, Octubre 1988, ISSN 0001-0782. `https://doi.org/10.1145/63039.63043`.

[38] Traoré, Mamadou: *SimStudio: a Next Generation Modeling and Simulation Framework*. página 67, Enero 2008.

[39] Van Tendeloo, Yentl y Hans Vangheluwe: *PythonPDEVS: A Distributed Parallel DEVS Simulator*. DEVS '15, San Diego, CA, USA, 2015. Society for Computer Simulation International, ISBN 9781510801059.

[40] Vangheluwe, Hans. `http://msdl.cs.mcgill.ca/projects/PythonPDEVS/`.

[41] Vieira, Marco, Henrique Madeira, Kai Sachs y Samuel Kounev: *Resilience benchmarking*. En *Resilience assessment and evaluation of computing systems*, páginas 283–301. Springer, 2012.

[42] Wainer, G.: *Developing a software toolkit for urban traffic modeling.* Softw., Pract. Exper., 37:1377–1404, Noviembre 2007.

[43] Wainer, G., Sergio Daicz, Alejandro Troccoli y Planta I: *Experiences In Modeling And Simulation Of Computer Architectures In Devs.* SIMULATION: Transactions of The Society for Modeling and Simulation International, 18, Febrero 2002.

[44] Wainer, Gabriel, Ezequiel Glinsky y Marcelo Gutierrez-Alcaraz: *Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark.* SIMULATION, 87(7):555–580, Enero 2011. https://doi.org/10.1177/0037549710395649.

[45] Wainer, Gabriel A.: *CD++: a toolkit to define discrete-event models.* Software, Practice and Experience. Wiley, 32(3):1261–1306, November 2002. http://cell-devs.sce.carleton.ca/publications/2002/Wai02.

[46] Weicker, Reinhold P.: *Dhrystone: A Synthetic Systems Programming Benchmark.* Commun. ACM, 27(10):1013–1030, Octubre 1984, ISSN 0001-0782. https://doi.org/10.1145/358274.358283.

[47] Wymore, A. Wayne: *A mathematical theory of systems engineering: The elements.* John Wiley and Sons, 1967.

[48] Yentl: *yentl/pythonpdevs.* https://msdl.uantwerpen.be/git/yentl/PythonPDEVS.

[49] Zeigler, Bernard: *Theory of modeling and simulation : discrete event and iterative system computational foundations.* Academic Press, an imprint of Elsevier, London, United Kingdom, 2019, ISBN 978-0-12-813370-5.

[50] Zeigler, Bernard P. y Alexander Muzy: *From Discrete Event Simulation to Discrete Event Specified Systems (DEVS).* IFAC-PapersOnLine, 50(1):3039–3044, 2017, ISSN 2405-8963. https://www.sciencedirect.com/science/article/pii/S2405896317310601, 20th IFAC World Congress.