Escuela Técnica Superior de Ingeniería Informática
Departamento de Tecnología Electrónica

Tesis doctoral

Implementación sobre hardware reconfigurable de una arquitectura no determinista, paralela y distribuida de alto rendimiento, basada en modelos de computación con membranas

Realizada por
Juan Quirós Carmona
Ingeniero en Informática

Dirigida por

Dr. Alejandro Millán Calderón
Prof. Contratado Doctor

Dr. Julián Viejo Cortés
Prof. Contratado Doctor Interino

Sevilla, 31 de octubre de 2015

Resumen

En este documento se presenta el trabajo de tesis doctoral realizado dentro del Programa de Doctorado "Informática Industrial" del Departamento de Tecnología Electrónica de la Universidad de Sevilla. Recoge la investigación centrada en el desarrollo de una implementación en *hardware* reconfigurable, FPGA, de modelos de computación basados en membranas, también denominados sistemas P. Estos sistemas, de inspiración biológica, son de reciente creación, y tienen aplicaciones directas en procesos de simulación, especialmente de sistemas y procesos biológicos. Se engloban dentro de la computación natural, y se trata de modelos paralelos maximales orientados a máquinas. Este hecho supone un desafío en el desarrollo de implementaciones *hardware*, ya que es precisa la generación de un diseño diferente para cada problema, incluso para cada instancia. Como consecuencia directa, es necesario el desarrollo de una arquitectura *hardware* dedicada parametrizada, junto con un desarrollo *software*, que analice los sistemas de entrada y, en base a sus características, construya un diseño sintetizable dedicado para esa instancia concreta. Además, al ser la disciplina de reciente creación, existen distintos tipos de sistemas P, por lo que es preciso un análisis previo, seguido de una selección, con el propósito de implementar el mayor subconjunto posible de los mismos.

Agradecimientos

Me gustaría agradecer su ayuda a todas aquellas personas que, de un modo u otro, han contribuido con su granito de arena a la conclusión de este trabajo. Empezando por mis familiares, especialmente mis padres Juan y Mª del Carmen, mi hermana Setefilla, mi novia Gema y mis abuelos. También a todos mis amigos, entre ellos Josemi, Eli, Rocío, Fer y Juanma. No puedo olvidarme de todos mis compañeros del Departamente de Tecnología Electrónica, especialmente a mis directores de tesis, Alejandro y Julián, a Manolo por todo su apoyo, y al resto de mis compañeros del Grupo de Investigación y Desarrollo Digital, ya que sin su ayuda no estaríais leyendo estas líneas. Por último, también dar las gracias al apoyo del Grupo de Computación Natural de la Universidad de Sevilla por su colaboración.

Este trabajo ha sido parcialmente financiado por el Ministerio de Ciencia e Innovación del Gobierno de España a través del proyecto TEC2011-27936 (HIPERSYS), el proyecto TEC2007-61802 (HIPER), por el Fondo Europeo de Desarrollo Regional (FEDER) y por el Ministerio de Educación, Cultura y Deporte a través de la beca FPU AP2009-3625.

Índice general

Resumen **1**

Agradecimientos **3**

1. Introducción **17**

1.1. Tecnologías de implementación 18

1.1.1. Sistema de computación secuencial 19

1.1.2. Sistemas de computación paralelos basados en *software* . . . 19

1.1.3. Sistemas de computación paralelos basados en *hardware* . . 21

1.2. Implementaciones de modelos de computación con membranas . . . 24

1.3. Objetivos y estructura de la tesis 26

I Análisis del estado del arte **29**

2. La computación con membranas **31**

2.1. Computación con membranas 32

2.1.1. Definición de un sistema de membranas 37

2.1.2. Modelo de computación con membranas. Un ejemplo 43

2.1.3. Aplicaciones de los sistemas P 47

2.2. Implementaciones actuales 51

2.2.1. Requisitos de diseño 51

2.2.2. Trabajos de implementación *software* 52

2.2.3. Trabajos de implementación *hardware* 58

2.3. Conclusiones 70

II Desarrollo de la arquitectura Almond PS y del *software* de generación **73**

3. Arquitectura Almond PS **75**

3.1. Evolución 76
3.1.1. La raíz del problema 76
3.1.2. Resolución de otros autores 77
3.1.3. Gramáticas libres de contexto como generadores de series formales de potencias 81
3.2. Fundamentos teóricos. Parte formal 86
3.3. Sistemas P aceptados por la arquitectura Almond PS 95
3.4. Implementación *hardware* 99
3.4.1. Bloque de persistencia 101
3.4.2. Bloque de cómputo de N_{r_i} 105
3.4.3. Bloque de asignación 107
3.4.4. Bloque de aplicación 117
3.4.5. Bloque de entrada/salida 119
3.4.6. Bloque de control 121
3.5. Implementando nuevos sistemas P 124
3.6. Conclusiones 127

4. *Software* de generación de sistemas P **129**

4.1. Procesamiento de los sistemas de entrada 129
4.2. Componentes de generación *hardware* 131
4.2.1. Estructuración de los módulos específicos de generación . . . 133
4.2.2. Elementos generadores de Almond PS 135
4.3. Mejora del *software* de generación con *Model-Driven Engineering* (MDE) 137
4.3.1. Breves nociones de MDE 138
4.3.2. Generación de componentes *hardware* con MDE 141
4.3.3. Aplicación de MDE en la generación de pruebas unitarias de módulos 143
4.4. Conclusiones 151

III Escenarios de verificación, pruebas y resultados 153

5. Análisis de Almond PS. Pruebas y resultados 155

5.1. Sistemas P de prueba . . . 155
5.2. Resultados funcionales . . . 160
5.2.1. Análisis de las configuraciones finales . . . 160
5.2.2. Análisis de las transiciones necesarias para alcanzar la condición de parada total . . . 183
5.2.3. Análisis de la distribución en el espacio de solución de las configuraciones finales . . . 187
5.3. Recursos *hardware* . . . 190
5.3.1. Recursos *hardware* según dependencias entre reglas . . . 191
5.3.2. Recursos *hardware* según complejidad del sistema de entrada 192
5.3.3. Recursos *hardware* según cardinalidades máximas . . . 195
5.3.4. Recursos *hardware* según modo de generación *software* . . . 198
5.3.5. Recursos *hardware* según tecnologías de *Field Programmable Gate Array* (FPGA) . . . 200
5.4. Análisis de rendimiento . . . 202
5.5. Conclusiones . . . 203

Conclusiones finales 207

Publicaciones 209

Abreviaturas 211

Bibliografía 217

Índice de tablas

3.1. Interfaz del Bloque de Entrada/Salida. 119
3.2. Códigos de estado del sistema ofrecidos por el Bloque de Control. . 120
3.3. Códigos de control generados por el Bloque de Entrada/Salida. . . . 120

Índice de figuras

1.1. Estructura por bloques de las FPGA de la marca Xilinx. 23

2.1. Estructura de una célula eucariota animal. 33
2.2. Representación de una estructura jerárquica de membranas. 34
2.3. Representación gráfica de la ejecución del sistema P de ejemplo. . . 46
2.4. Esquema de interconexión propuesto por Petreska. 59
2.5. Esquema de un *membrane hardware component*. 60
2.6. Esquema de la arquitectura *Reconfig-P* α. 66
2.7. Esquema de la arquitectura *Reconfig-P* β 69

3.1. Diagrama de actividad UML que muestra la división en funcionalidad de la ejecución de un sistema P. 78
3.2. Visión general de la arquitectura Almond PS 102
3.3. Esquemático del bloque de persistencia. 104
3.4. Esquema del bloque de cómputo de N_{r_i}. 106
3.5. Detalle de la arquitectura del bloque de asignación. 109
3.6. Esquema del bloque de aplicación. 118
3.7. Diagrama de estados simplificado de la unidad de control. 122

4.1. Diagrama UML simplificado de la solución *software* de generación. . 132
4.2. Diagrama UML simplificado de la estructura base de la herramienta de generación de código. 134
4.3. Diagrama *Unified Modeling Language* (UML) simplificado de la solución *software* de generación de código de la plataforma Almond PS. 135

4.4. Modelos y transformaciones entre modelos de un *framework* de simulación de sistemas basado en MDE. Las flechas representan transformaciones entre modelos. 140
4.5. Modelo tinyVHDL empleado en el *software*. 142
4.6. Diagrama simplificado de los tres pasos de la metodología de pruebas. 147

5.1. Ejemplo de sistemas P de referencia para un tamaño de 10 reglas. . 157
5.2. Ejecución, con Almond PS, de sistemas P de tamaño 20 reglas y dependencia circular. Configuración de parada. 162
5.3. Distancia de los resultados obtenidos con Almond PS respecto a los resultados más probables. Dependencia circular 163
5.4. Ejecución, con Almond PS, de sistemas P de tamaño 20 reglas y dependencia circular-2. Configuración de parada. 165
5.5. Multiplicidad media contextualizada de las distintas ejecuciones con Almond PS de sistemas P con dependencia circular-2. 166
5.6. Ejecución, con Almond PS, de sistemas P de tamaño 20 reglas y dependencia lineal. Configuración de parada. 167
5.7. Multiplicidad media contextualizada de las distintas ejecuciones con Almond PS de sistemas P con dependencia lineal. 168
5.8. Ejecución, con Almond PS, de sistemas P de tamaño 20 reglas y dependencia opuesta. Configuración de parada. 170
5.9. Valores medios y máximos de las distancias de las distintas ejecuciones de Almond PS en comparación con los resultados más probables. 171
5.10. Ejecución, con Almond PS y P-Lingua, de sistemas P de tamaño 20 reglas y dependencia circular. Configuración de parada. 174
5.11. Distancia de los resultados obtenidos con Almond PS y P-Lingua respecto a los resultados más probables, para un tamaño de 20 reglas. Dependencia circular 175
5.12. Distancia de los resultados obtenidos con Almond PS y P-Lingua respecto a los resultados más probables, para distintos tamaños. Dependencia circular 176
5.13. Ejecución, con Almond PS y P-Lingua, de sistemas P de tamaño 20 reglas y dependencia circular-2. Configuración de parada. 177

5.14. Multiplicidad media contextualizada de las distintas ejecuciones con Almond PS y P-Lingua de sistemas P con dependencia circular-2. . 178
5.15. Ejecución, con Almond PS y P-Lingua, de sistemas P de tamaño 20 reglas y dependencia lineal. Configuración de parada. 180
5.16. Multiplicidad media contextualizada de las distintas ejecuciones con Almond PS y P-Lingua de sistemas P con dependencia lineal. . . . 181
5.17. Ejecución, con Almond PS y P-Lingua, de sistemas P de tamaño 20 reglas y dependencia opuesta. Configuración de parada. 182
5.18. Valores medios y máximos de las distancias de las distintas ejecuciones de Almond PS y P-Lingua en comparación con los resultados más probables. 183
5.19. Número de transiciones para alcanzar una condición de parada total. Comparación de Almond PS y P-Lingua. Modo de dependencia circular-2 185
5.20. Número de transiciones para alcanzar una condición de parada total. Comparación de Almond PS y P-Lingua. Modo de dependencia lineal. 186
5.21. Número de configuraciones finales diferentes para tamaños de 10, 20, 30 y 50 reglas, de cada uno de los sistemas P de referencia. Se muestran los resultados obtenidos con Almond PS (a) y P-Lingua (b). 188
5.22. Análisis de recursos *hardware* consumidos según dependencias de reglas. 192
5.23. Recursos *hardware* requeridos por la implementación según la complejidad del sistema, medida en número de reglas. LUT usa el eje Y de la izquierda, mientras que BRAM y DSP usan el eje Y de la derecha. 193
5.24. Análisis de recursos *hardware* consumidos según tamaño de entrada. 194
5.25. Análisis de LUT por regla, LUT por *slice* y DSP por regla empleados, según tamaño de entrada. 195
5.26. Análisis de la distribución de recursos LUT entre lógica, rutado y memoria, según tamaño de entrada. 196
5.27. Análisis de recursos *hardware* consumidos según tamaño de bus de datos. 197

5.28. Análisis de LUT/*Slice* empleados según tamaño de bus de datos. . . 198
5.29. Análisis de recursos *hardware* consumidos por bloque de la arquitectura según tamaño de bus de datos. 199
5.30. Análisis de recursos *hardware* consumidos según modo de generación *software*. 200
5.31. Recursos *hardware* requeridos por la implementación según las tres últimas series de FPGA del fabricante Xilinx. LUT usa el eje Y de la izquierda, mientras que DSP usa el eje Y de la derecha. Los *Look-Up Table* (LUT) empleados por las tres familias son iguales, por lo que su representación está solapada. 201
5.32. Rendimiento de Almond PS. 203
5.33. Análisis temporal de Almond PS y comparación con P-Lingua. . . . 204

Índice de algoritmos

2.1. Algoritmo de Marcado 40
2.2. Algoritmo paso a paso 62
2.3. Algoritmo de aplicación con *benchmark* de minimalidad 63
2.4. Algoritmo de aplicación con *benchmark* de minimalidad 64
3.5. Algoritmo para obtener la función *Variant* 91
3.6. Algoritmia de alto nivel del bloque de asignación 110
3.7. Funcionalidad específica del bloque de asignación 111
3.8. Propagación izquierda 113
3.9. Valores Auxiliares de Seq. de Padovan 114
3.10. Propagación derecha 115
3.11. Valores de aplicabilidad con regla dependiente 116

Capítulo 1

Introducción

Aunque el estudio y resolución de problemas ha sido una constante a lo largo de la historia del ser humano, no fue hasta el siglo XX cuando se formalizó el concepto de procedimiento mecánico. Este es consecuencia de la búsqueda por parte de un grupo de matemáticos, entre los que destaca D. Hilbert, de un método universal con el que resolver cualquier problema matemático [Hilbert, 1918]. En la década de los 30, K. Gödel publica su Teoría de la Incompletitud [Gödel, 1931] que, junto con resultados posteriores, demuestra que es imposible lo pretendido por D. Hilbert.

A partir del trabajo de K. Gödel se originan los primeros modelos de computación (λ-cálculo de A. Church y S. Kleene [Church, 1936], funciones recursivas de K. Gödel [Gödel, 1931] y máquinas de Touring de A. Touring [Turing, 1937]). Estos modelos de computación sientan las bases para el desarrollo de las máquinas de computación teóricas, entre las que se encuentra la arquitectura de Von Neumann [Neumann, 1945], implementada en la gran mayoría de los ordenadores actuales.

Aunque los primeros ordenadores ofrecían una potencia computacional desconocida hasta la fecha, poseían serias limitaciones de memoria y velocidad de cómputo. De ese modo, el análisis de los recursos requeridos por los algoritmos pasa a cobrar especial relevancia, otorgando un enfoque distinto al estudio y resolubilidad práctica de problemas: la eficiencia de la solución es igual de importante que esta. En consecuencia, aparecen problemas computables que, dada la cantidad de recursos requeridos por los algoritmos que los resuelven, no son abordables en

la práctica con las máquinas del momento (ni actuales), esto es, son intratables. Desgraciadamente, algunos de estos son de interés práctico y los medios de los que disponemos son, claramente, insuficientes.

Es por ello que se empiezan a estudiar modelos de computación alternativos a los convencionales (máquinas de Touring, funciones recursivas y λ-cálculo, entre otros) que amplíen el número de problemas resolubles en la práctica. Dentro de estos nuevos modelos se encuentran aquellos englobados bajo el término de computación natural [Rozenberg et al., 2012], caracterizados por estar inspirados en procesos de la naturaleza. Ejemplos de estos son los algoritmos genéticos, inspirados en la evolución y selección natural; las redes neuronales, basadas en el sistema nervioso; la computación molecular, que emplea moléculas orgánicas como el Ácido desoxirribonucleico (ADN); y la computación con membranas [Păun, 2000], inspirada en las células de organismos vivos.

De entre todos los anteriores, la computación con membranas, o sistemas P en referencia al inventor de la línea de investigación, G. Pâun, constituye la disciplina más reciente. Aunque en la actualidad no es posible aprovechar sus beneficios en materia de potencia de cómputo al no existir una implementación directa, se ha demostrado su utilidad en la resolución de múltiples problemas, especialmente aquellos de simulación de sistemas complejos. Esta aplicación práctica, motiva que sea de especial interés la implementación eficiente de estos modelos.

Este capítulo presenta, en primer lugar, las tecnologías existentes en la actualidad, candidatas a ser escogidas para el desarrollo de sistemas P. En este sentido, la tecnología *Field Programmable Gate Array* (FPGA) se introduce con mayor detalle que el resto, al ser la empleada en el trabajo presentado. Como segundo y último punto, se detallan los objetivos que persigue el trabajo de investigación realizado por el doctorando, así como la estructura de este documento.

1.1. Tecnologías de implementación

En la actualidad existe una gran número de sistemas de computación disponibles, permitiendo elegir el más conveniente en base a los requisitos de diseño del problema. Así, los últimos años han estado marcados por la irrupción de sistemas de procesamiento paralelos, especialmente en la electrónica de consumo. El abara-

tamiento de estos dispositivos, ha permitido obtener mayor potencia computacional a menor coste, haciendo viables muchos desarrollos, anteriormente prohibitivos. Es por ello que se presentan, a continuación, los distintos sistemas de computación disponibles en la actualidad.

1.1.1. Sistema de computación secuencial

Se caracteriza por la ejecución de un *software* sobre una arquitectura *hardware* compuesta por un procesador de propósito general o específico. El modo de ejecución es secuencial, esto es, finaliza una única instrucción por unidad de tiempo. Aunque la potencia de este tipo de arquitecturas puede ser incrementada con técnicas de diseño entre las que destaca el *pipeline*, actualmente depende notablemente de los avances en tecnología electrónica y en los procesos de fabricación. Su principal ventaja reside en ser la plataforma más simple desde el punto de vista del diseño, originando implementaciones extensibles y modulares; no obstante, es la de menor potencia de cómputo. Esta limitada capacidad de evolución y las necesidades de computación han ocasionado que en los últimos años se opte por sistemas de computación multihilo, incluso en dispositivos móviles.

1.1.2. Sistemas de computación paralelos basados en *software*

En este tipo de sistemas de computación, el *software* es ejecutado sobre uno o más núcleos o procesadores de modo paralelo, por lo que dos o más instrucciones finalizan por unidad de tiempo. En función de la arquitectura *hardware* se puede distinguir entre sistemas distribuidos y no distribuidos.

1.1.2.1. Sistemas de computación basados en *software* no distribuido

En los sistemas paralelos no distribuidos las unidades de procesamiento son locales, generalmente localizadas en el mismo integrado. Debido a la limitación de las arquitecturas secuenciales, este tipo de plataformas han experimentado una gran difusión en los últimos años, copando el mercado de consumo casi por completo. Dentro de este grupo se distinguen los procesadores multihilo de propósito

general y las *Graphic Processor Unit* (GPU).

Los procesadores multihilo son el resultado de la evolución de los procesadores secuenciales, replicando en el mismo integrado varios núcleos de este tipo. El procesamiento es organizado en hilos, compartiendo recursos tales como espacios de direcciones o datos. El paralelismo se logra al asignar hilos a los distintos núcleos que componen el procesador, aumentando de forma significativa la cantidad de procesado por unidad de tiempo. No obstante, el multiprocesamiento supone un trabajo extra de implementación, originándose problemas de sincronización y de exclusión mutua, que requieren el diseño de algoritmos específicos o adaptación de los existentes en el mejor de los casos. Además, el resto de recursos no es multiplicado por el mismo coeficiente, por lo que pueden originarse cuellos de botella. En conclusión, presentan una potencia de cómputo superior a los anteriores a costa de requerir mayor esfuerzo en su diseño, una extensibilidad similar y una mejor escalabilidad, aunque limitada por el número de núcleos contenidos en el procesador.

Las GPU son unidades de procesado específicas destinadas a la generación de gráficos. Son, por lo tanto, componentes optimizados para una tarea concreta, que se reduce al cálculo de vértices y píxeles. Las aplicaciones gráficas se caracterizan por el predominio de operaciones en coma flotante y un alto grado de paralelismo inherente, posibilitando el uso de unidades de cálculo completamente independientes. Desde sus inicios, en su arquitectura las unidades funcionales se organizaban en dos tipos: aquellas destinadas al procesamiento de vértices y las destinadas al procesamiento de píxeles. En 2006, NVIDIA[1] introduce en el mercado la arquitectura Tesla, que unifica ambas unidades funcionales y las extiende [Lindholm et al., 2008; Sanders and Kandrot, 2011], presentando una matriz escalable de procesadores [NVIDIA, 2011]. La adopción de esta nueva plataforma y el lanzamiento en 2007 de *Compute Unified Device Architecture* (CUDA), un conjunto de herramientas de desarrollo que permiten la implementación de algoritmos en sus GPU, empleando una variación del lenguaje C, convirtieron a las GPU en sistemas masivamente paralelos de bajo coste [NVIDIA, 2010]. La arquitectura Tesla está basada en el modelo *Single Program Multiple Data* (SPMD), donde una instrucción *Single Instruction Multi-Thread*, SIMT, es ejecutada paralelamente sobre un

[1]NVIDIA: http://www.nvidia.com

conjunto de datos. En consecuencia, las GPU son adecuadas para algoritmos de gran paralelismo de datos o paralelismo de grano fino (*fine-grained*), siendo su uso desaconsejado para aplicaciones que requieran un paralelismo de grano grueso (*coarse-grained*) o a nivel de procesos, o aquellas que precisen gran cantidad de procesos de comunicación o acceso a memoria.

1.1.2.2. Sistemas de computación basados en *software* distribuido

Los sistemas de computación paralelos y distribuidos están compuestos por varios nodos o *hosts* comunicados a través de una red de comunicaciones. Gráficamente pueden ser descritos como un grafo en el que las aristas representan los enlaces de comunicación entre equipos, representados por los nodos. Este tipo de sistemas se definen por la topología de la red de interconexión y por los equipos, que pueden ser heterogéneos. Generalmente requieren un importante esfuerzo de implementación, debido a los problemas de sincronización y comunicación, que dependen de las características de la red empleada y del diseño e implementación de la aplicación. Tanto es así que el factor que relaciona el tiempo de comunicación y el de cómputo adquiere especial relevancia y determina la utilidad práctica del sistema diseñado. Los sistemas distribuidos son adecuados para la implementación de algoritmos con paralelismo de grano grueso, en el que las comunicaciones son muy reducidas en comparación con el tiempo de computación.

1.1.3. Sistemas de computación paralelos basados en *hardware*

Los sistemas *hardware* se caracterizan por el paralelismo y la concurrencia. Ofrecen una potencia superior a las implementaciones *software*, debido a su optimización para resolver un problema concreto, eliminación de la capa *software* y su paralelismo implícito. No obstante, es la alternativa que supone un mayor esfuerzo de desarrollo, y el área aparece como un recurso crítico que limita la complejidad de los algoritmos a implementar. Considerando los dispositivos lógicos programables, existen dos alternativas *hardware*: los *Application-Specific Integrated Circuit* (ASIC) y las FPGA. Con respecto a los primeros, son circuitos integrados hechos a medida para una aplicación específica. Sus principales ventajas son el ren-

dimiento, consumo, coste y frecuencia de reloj; su principal desventaja consiste en que una vez fabricados no es posible modificarlos. Por el contrario, la tecnología FPGA permite reconfigurar el *hardware*, encontrándose en un punto intermedio entre el *software* y los ASIC.

1.1.3.1. La tecnología *Field Programmable Gate Array*

La tecnología FPGA fue inventada a mediados de los años ochenta por la compañía XILINX[2] y se considera la evolución de al tecnología *Complex Programmable Logic Device* (CPLD) [Wilson, 2007]. Las FPGA son dispositivos semiconductores programables basados en *Configurable Logic Blocks* (CLB) conectados entre sí empleando una red de interconexión configurable y con el exterior a través de unos bloques de entrada/salida denominados *Input/Output Blocks* (IOB) (Fig. 1.1). Los CLB constituyen la unidad reprogramable básica y están formados por varias celdas lógicas denominadas *Slices*. Los *Slices* se componen, a su vez, de varias tablas de búsqueda (*Look-Up Table*, LUT) de un determinado número de entradas que permiten definir cualquier función lógica de aridad igual al número de estas, un sumador completo que incluye lógica de acarreo, multiplexores y biestables (*Flip Flops*, FF). Además, disponen de bloques de memoria volátil denominados *Block RAM* (BRAM) y bloques de gestión digital del reloj (*Digital Clock Manager*, DCM), ambos distribuidos por su área. Con este tipo de dispositivos es posible implementar cualquier circuito digital, siendo las únicas restricciones los recursos *hardware* disponibles en el chip y la frecuencia máxima de operación que se pueda alcanzar.

Las principales ventajas de las FPGA derivan de su característica fundamental: la reconfigurabilidad[3]. Esta reduce considerablemente los tiempos de desarrollo de un circuito digital, ya que simplifica y determina el ciclo de diseño y posibilita la reconfiguración del dispositivo un número prácticamente ilimitado de veces, incluso de forma remota y parcial en tiempo de ejecución. Estas características les otorgan una gran flexibilidad, convirtiéndolas en una destacada herramienta para labores de investigación y/o de prototipado. Como aspectos negativos, en compara-

[2] XILINX: `http://www.xilinx.com`

[3] Existen FPGA que incorporan una tecnología de memoria de programación basada en fusibles permitiendo configurar el dispositivo una única vez.

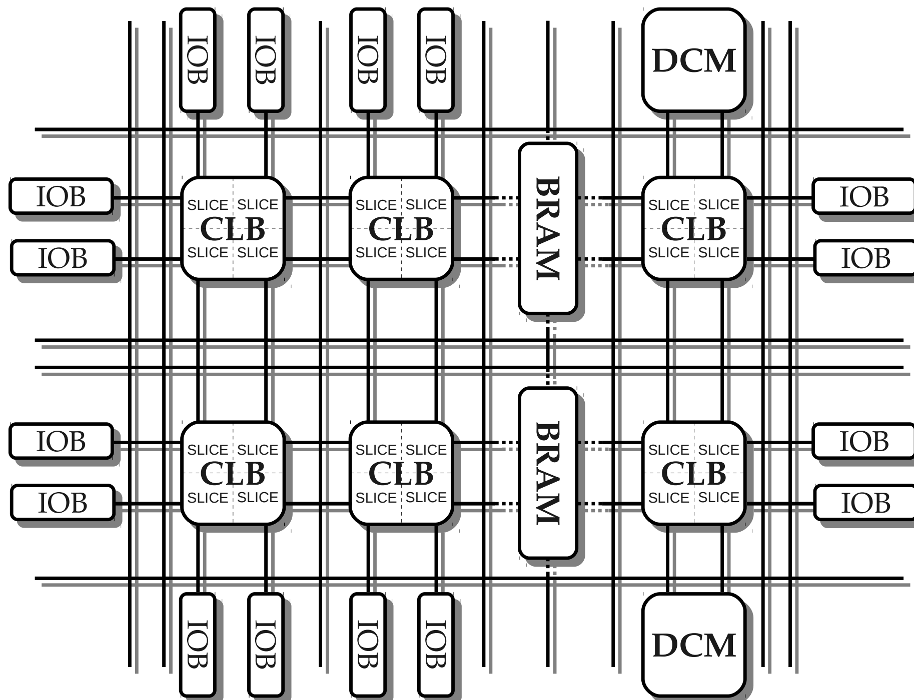


Fig. 1.1: Estructura por bloques de las FPGA de la marca XILINX.

ción con los ASIC, cabe citar que estos últimos permiten una mayor especificidad del *hardware* a cambio de perder parte de la flexibilidad proporcionada por las FPGA, consiguiendo una reducción de coste (para grandes tiradas) y de consumo y aumentando la frecuencia máxima de operación.

Actualmente existen varios fabricantes de FPGA. Entre ellos destacan XILINX, ALTERA[4], LATTICE SEMICONDUCTOR[5] y ACTEL[6], siendo los dos primeros los líderes del sector.

[4] ALTERA: `http://www.altera.com`
[5] LATTICE SEMICONDUCTOR: `http://www.latticesemi.com/`
[6] ACTEL: `http://www.microsemi.com/`

1.2. Implementaciones de modelos de computación con membranas

El desarrollo de implementaciones de sistemas P tuvo sus inicios poco después del nacimiento de la disciplina. De ese modo, las primeras implementaciones se trataban de versiones *software* secuenciales. Estas versiones, a pesar de resultar las más fáciles de desarrollar, extender y mantener, pronto evidenciaron una falta de potencia de cómputo, que las limitaban a tareas de demostración y asistencia durante el diseño de sistemas.

El siguiente paso fue natural, emplear tecnologías que ofrecieran una mayor potencia de cómputo, a través de su paralelismo inherente. Así, las implementaciones *software* paralelas atrajeron el interés en el área. En este sentido, se han desarrollado simuladores que hacen uso del paralelismo en procesadores de propósito general, a través del empleo de varias máquinas o del uso de aquellos con más de un núcleo de procesamiento. Sin embargo, la potencia de cómputo ofrecida por estas tecnologías no es suficiente.

Al analizar un sistema de computación con membranas, lo primero que llama la atención es la simplicidad de las operaciones que son requeridas para su ejecución. El gran escollo se presenta en el doble paralelismo inherente de estos sistemas. A grandes rasgos, los sistemas P están compuestos por conjuntos de unidades que transforman recursos existentes en otros nuevos en cada paso de computación. Un sistema está compuesto por cientos de este tipo de unidades, que compiten entre sí por los recursos existentes. Es, por lo tanto, un problema de distribución de recursos, agravado por las fuertes restricciones que se imponen en esta competición, desde la necesidad de agotar solo un determinado número de estos recursos, hasta el máximo posible, y todo ello de un modo no determinista, que hace necesario resolver, a su vez, un problema de gran esfuerzo computacional, el de la distribución de recursos, en cada paso de computación.

Es evidente que el desarrollo de simuladores o implementaciones que empleen tecnología electrónica convencional, no podrá ofrecer el mismo rendimiento que aquellos desarrollados, hipotéticamente, con tecnología orgánica. Es por ello que, actualmente, el objetivo del área es la resolución de aquellos problemas de interés

práctico en un tiempo aceptable. En este sentido, las últimas líneas de investigación se centran en tecnologías donde existe un alto grado de paralelismo: desarrollo *software* sobre GPU y *hardware* sobre FPGA.

Son obvias las ventajas en potencia de cómputo que supone un dispositivo *hardware* especialmente diseñado para la aplicación en este ámbito. De ese modo, la tecnología FPGA se postula como una de las más adecuadas, dado su equilibrio (a nivel *hardware*) entre potencia y flexibilidad. En este contexto, existen dos trabajos previos, el desarrollado por B. Petreska y C. Teuscher [Petreska and Teuscher, 2004], que presenta a las FPGA como dispositivos válidos para este fin, y el llevado a cabo por V. Nguyen [Nguyen, 2010], donde se describe una solución para un subconjunto de sistemas P, centrados en aquellos deterministas, aunque presenta las primeras líneas para los no deterministas.

Un inconveniente añadido para una implementación *hardware* de sistemas de computación con membranas, es el hecho de ser modelos orientados a máquina. Es por ello que para cada problema es preciso contruir una máquina distinta, por lo que el desarrollo de una implementación se dirige al diseño de una arquitectura en base a la cual se construirán los distintos sistemas P en función de los problemas de entrada. Además, es preciso acompañar a este desarrollo de un *software* de generación de instancias concretas para, de ese modo, dotar de utilidad al trabajo realizado. En este sentido, Nguyen ha diseñado una solución para la generación de código. No obstante, esta se encuentra demasiado acoplada con su arquitectura, siendo conveniente el desarrollo de una herramienta que permita la modificación del *hardware* sin que requiera conocimientos exhaustivos de este.

En consecuencia, el siguiente reto consiste en el desarrollo de una implementación que, además de destacar por su potencia computacional, y mantener una flexibilidad y escalabilidad adecuadas, amplíe el espectro de sistemas P admitidos, centrándose en aquellos no deterministas, que son los de mayor interés práctico. Además, los desarrollos de Nguyen y Petreska toman como referencia de diseño la estructura de los sistemas P. En este sentido, se busca focalizar el diseño en las propias características de la tecnología FPGA, y no en la de los sistemas P, con el objetivo de obtener sistemas con el mayor rendimiento posible. Como elemento adicional, también es necesario el desarrollo de una herramienta *software* de generación, con un nivel de acoplamiento inferior a la existente, que facilite la

modificación del *hardware* desarrollado.

1.3. Objetivos y estructura de la tesis

El objetivo general de este trabajo de tesis consiste en el desarrollo de una implementación *hardware* de sistemas P. Para ello, es necesario llevar a cabo, previamente, un análisis de los métodos, algoritmos, arquitecturas y soluciones actuales. Dadas las características del modelo computacional objeto de este trabajo, se fija como objetivo el desarrollo de una arquitectura genérica, junto con un procedimiento automático de generación, con el fin de cubrir el conjunto de las necesidades que requieren los sistemas P.

Así, se presenta la arquitectura Almond *P System* (Almond PS). Se trata de una arquitectura modular, en la que se encapsulan los distintos componentes necesarios para la ejecución de los sistemas P, con el propósito de ofrecer la máxima flexibilidad y escalabilidad posibles. Además, a diferencia de otras implementaciones actuales, únicamente es preciso tener en consideración el grafo de dependencias entre sus reglas a la hora de seleccionar los sistemas compatibles. Del mismo modo, también se presenta un *software* de generación de implementaciones de la arquitectura Almond PS, que constituye un primer paso hacia un *framework* de generación de código *Hardware Description Language* (HDL).

A continuación, se detallan los objetivos concretos del trabajo de tesis presentado en este documento:

1. Análisis del modelo de computación con membranas, abstracción de aspectos estructurales y funcionales y establecimiento de requisitos.

2. Análisis de implementaciones y algoritmos de implementación actuales, incluyendo *hardware* y *sofware*.

3. Implementación de una arquitectura de computación (Almond PS) que permita la simulación de sistemas P, en base a los requisitos establecidos.

4. Diseño de una herramienta *software* de simulación, que permita la generación automática de instancias concretas de dispositivos bajo la arquitectura Almond PS, y posterior desarrollo de los elementos funcionales básicos.

5. Validación de los resultados obtenidos mediante una extensa batería de pruebas, que permita el análisis del conjunto de factores que influyen en los sistemas generados. Incluye el establecimiento previo de los escenarios de prueba, así como del desarrollo de las herramientas necesarias.

Como punto final, se describe como se organiza el resto del documento:

- Parte *I*. Está formada por el capítulo 2, y satisface los objetivos 1 y 2 (parcialmente). En este capítulo se presenta la computación con membranas o sistemas P, con una introducción informal, seguida de una definición formal, enfocada a los conceptos necesarios para la comprensión del desarrollo de la arquitectura Almond PS. Esta introducción es completada con la descripción funcional de un ejemplo, y finaliza con la enumeración de las principales aplicaciones actuales, o en un futuro cercano, de los sistemas P. Una vez aportados los conocimientos previos necesarios, se analizan las implementaciones más destacadas de este modelo, haciendo énfasis en aquellos trabajos más próximos a la arquitectura Almond PS.

- Parte *II*. Los capítulos 3 y 4 le dan cuerpo a esta parte, respondiendo a los objetivos 2 (parcialmente), 3 y 4. El capítulo 3 alberga la descripción de la arquitectura Almond PS. Así, en primer lugar realiza un análisis de las propuesta de otros autores a los problemas de algoritmia observados en los sistemas P. Seguidamente, se introducen los aspectos formales considerados en el diseño de la arquitectura y, por último, esta se describe detalladamente. Respecto al capítulo 4, muestra el desarrollo *software* que permite la generación automática de instancias concretas basadas en la arquitectura Almond PS, así como la metodología empleada.

- Parte *III*. En esta sección, formada por los capítulos 5 y 6, se da respuesta al último objetivo, el número 5 siguiendo la numeración anterior. Así, en primer lugar se describen los escenarios de prueba, seguidos por las herramientas empleadas para la adquisición y análisis de estos resultados. Por último, se analizan los resultados obtenidos, de los cuales se extraerán las conclusiones más relevantes de este trabajo, descritas en el capítulo 6.

Parte I

Análisis del estado del arte

Capítulo 2

La computación con membranas

Tal y como se ha comentado en el capítulo de introducción, la computación con membranas es una disciplina de reciente creación, que se enmarca dentro de los modelos de computación alternativos. Aunque el actual desarrollo científico no ha permitido plasmar este modelo de computación en dispositivos reales, genera un alto grado de interés debido a las ventajas que presenta en diferentes áreas, especialmente aquellas relacionadas con la simulación de procesos orgánicos e inorgánicos.

Este capítulo persigue ofrecer una visión general del estado de esta disciplina, no únicamente desde un punto de vista teórico, sino también de las distintas implementaciones existentes, sin olvidar los posibles campos de aplicación en los que resulta de utilidad a día de hoy o en un futuro próximo.

Con este propósito, el capítulo se estructura en tres apartados. El primero de ellos está dedicada a la descripción de la computación con membranas, desde un punto de vista teórico. Esta sección empieza con una introducción y definición informal a los sistemas P, enfocada para aquellos lectores para los cuales este documento constituye su primer contacto con este modelo de computación. Seguidamente, se ofrece una definición formal, centrada en aquellos conceptos o desarrollos teóricos alineados con el propósito de la investigación llevada a cabo por el doctorando. A continuación se muestra un ejemplo de un sistema P, al tiempo que se detalla una computación, con la finalidad de afianzar los conceptos introducidos previamente. Como último punto, se enumeran algunas aplicaciones

de interés.

El segundo apartado está dedicado a las implementaciones actuales. Así, se enumeran algunos requisitos de diseño que permitan fijar un criterio de comparación de los diferentes trabajos existentes, los cuales se clasifican en base a los dispositivos que los soportan. Este bloque continúa con la descripción de estos trabajos, ofreciendo un mayor nivel de detalle en aquellas implementaciones que guardan mayor relación con la implementación presentada en este documento.

Tras finalizar el análisis teórico de la computación con membranas y de las implementaciones existentes, se comentan las conclusiones más importantes de este capítulo.

2.1. Computación con membranas

La computación con membranas es un modelo de computación alternativo introducido por G. Păun en 1998 [Păun, 2000]. Está inspirado en el funcionamiento de las células eucariotas y se trata de un modelo paralelo distribuido y computacionalmente universal (capaz de resolver los mismos problemas que las máquinas de Turing. A diferencia de la gran mayoría de modelos utilizados en la actualidad, es un modelo orientado a máquinas, por lo que la implementación de un algoritmo no consiste en una secuencia finita de instrucciones, como sucede con un modelo orientado a programas, el más extendido, sino que se encuentra implícita en la descripción de una máquina.

Una célula (Fig. 2.1) se compone de: *(1)* una envoltura que le confiere entidad y que, en el caso de las células animales, se corresponde con una membrana celular; *(2)* el núcleo (en caso de ser una célula eucariota, en caso contrario se denomina célula procariota); y *(3)* el citoplasma, donde se localizan los orgánulos celulares *(4)*. Por lo tanto, en la célula se distinguen una serie de espacios o compartimentos (el propio citoplasma, los orgánulos y el núcleo celular) delimitados por membranas, existiendo una externa que le confiere individualidad. Las funciones de las membranas son las de actuar como delimitador y como canal de comunicación. En el interior de la región delimitada por la membrana existen moléculas. Estas pueden proceder del exterior, ser transportadas a este o a compartimentos internos a través de los canales de comunicación. Además, en estas regiones se producen reacciones

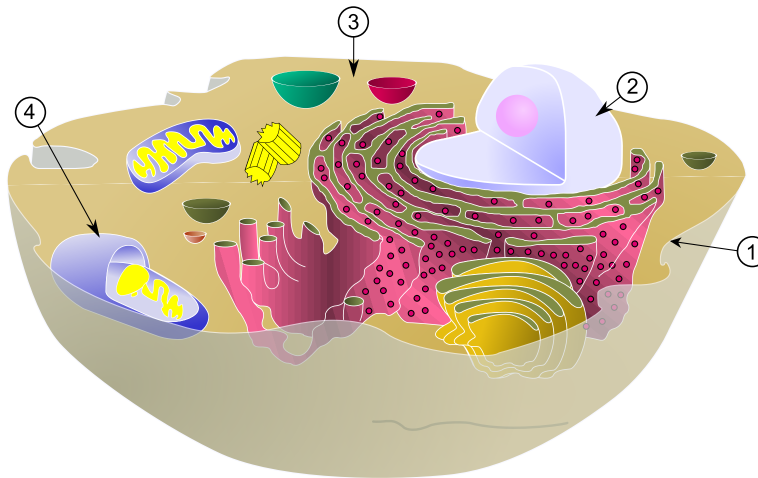


Fig. 2.1: Estructura de una célula eucariota animal. Se distingue (1) la membrana celular, (2) el núcleo celular, (3) el citoplasma, y (4) los orgánulos con sus membranas celulares. Versión modificada de la original creada por los usuarios MesserWoland y Szczepan1990 en *Wikipedia* y bajo licencia *Creative Commons*.

químicas, originando nuevas moléculas a partir de las existentes a una velocidad que puede ser modificada por los catalizadores.

En la actualidad existen gran cantidad de variantes de modelos de computación con membranas o sistemas P, siendo el sistema P transitivo, o sistema P de transición, el primigenio. Un sistema P de transición posee una estructura jerárquica de membranas (Fig. 2.2) en la que existe una externa, la raíz de la jerarquía, que le confiere entidad y que se denomina piel. De ese modo, una membrana define una región que puede contener más membranas o multiconjuntos de objetos o símbolos. Asociadas a las regiones se definen las reglas de evolución. Estas definen la semántica del sistema P, basada en la manipulación de objetos: transformación y comunicación entre membranas. Adicionalmente, también es posible disolver membranas, pasando todos sus objetos a la membrana padre. Las reglas se ejecutan de un modo síncrono paralelo maximal con prioridades, y en ocasiones no determinista. La componente probabilista intrínseca a las reacciones químicas y a los procesos de comunicación que se originan en el interior de las células es capturada con la definición de una relación de prioridad fuerte entre las reglas de evolución. Este concepto se describe con un ejemplo: si existen tres reglas, r_1, r_2 y r_3, de modo que r_1 y r_2 poseen mayor prioridad que r_3, esta última se ejecutará si y solo si no

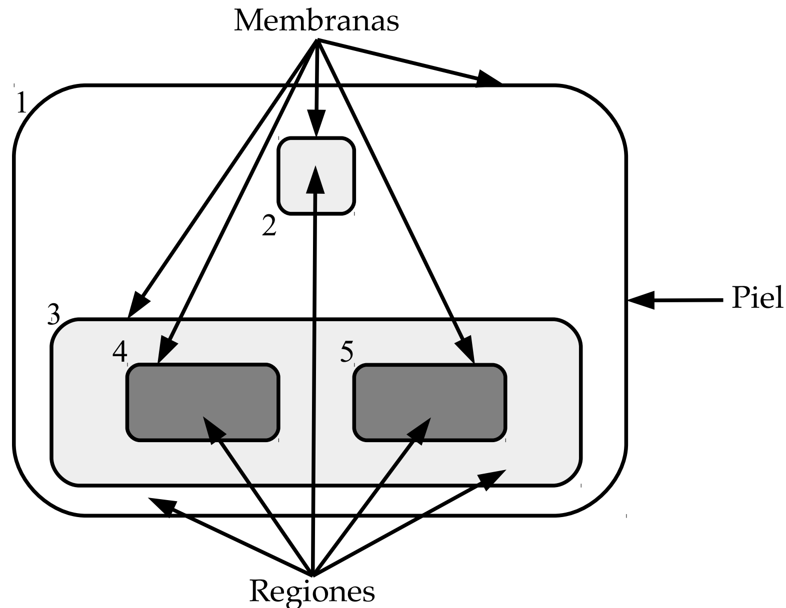


Fig. 2.2: Representación de una estructura jerárquica de membranas.

es posible ejecutar ni r_1 ni r_2, aunque haya objetos disponibles para que r_3 pueda ser ejecutada, e incluso los objetos implicados en r_3 sean distintos a los implicados en r_1 y r_2; para las reglas de igual prioridad o para las que no se establezca relación alguna, su ejecución será no determinista. Para otorgar simplicidad al modelo, se incorpora un reloj universal que rige la aplicabilidad de las reglas. Así, el sistema pasa de una configuración a otra en un ciclo de reloj o transición en el que se ejecutan todas las reglas aplicables, de acuerdo a las relaciones de prioridad y a los símbolos disponibles, un número determinado de veces. Estas se ejecutan de un modo paralelo y maximal. La maximalidad establece que el número de veces que es aplicada una regla en un ciclo de computación debe ser el máximo posible, consumiendo, a su vez, el máximo posible de objetos.

Desde su presentación, la disciplina ha experimentado un gran crecimiento, originándose una gran cantidad de ramificaciones que abarcan desde pequeñas modificaciones del sistema P inicial hasta el desarrollo de variedades nuevas, en las que perduran la inspiración biológica y el esquema de funcionamiento del original. Al ser un modelo computacional orientado a máquinas, el nacimiento de estas ramificaciones generalmente es motivado por la necesidad de resolver problemas

concretos. Ejemplos de estas extensiones son:

- Sistemas basados en la catálisis. Los sistemas P catalíticos (*catalytic P systems* [Păun, 2000, 2002]) se caracterizan por la presencia de catalizadores, considerados como objetos inalterables por las reglas y que son requeridos para su aplicación.

- Modificación de los procesos de comunicación entre membranas, contemplando la transferencia de objetos por concentración, sistemas P controlados por concentración (*concentration controlled P systems* [Păun, 2002; Dassow and Păun, 2001]), tal y como sucede biológicamente, asignando cargas a objetos y membranas [Păun, 2000, 2002] o modificando la permeabilidad de estas últimas [Păun, 2000, 2002].

- Eliminación de la membrana de salida, considerando la salida del sistema aquellos objetos enviados al entorno a través de la piel [Păun, 2002].

- Uso de promotores e inhibidores [Bottoni et al., 2002].

- Creación, disolución y división de membranas, dando lugar a los sistemas P con membranas activas (*P systems with active membranes* [Păun et al., 2010]).

- Sistemas P con simportadores/antiportadores (*P systems with symport/antiport* [Păun, 2002]), en los que dos objetos atraviesan la membrana simultáneamente en el mismo u opuesto sentido.

- Sistemas P con portadores (*P systems with carriers* [Păun, 2002; Păun and Păun, 2002]), en los que se diferencia entre objetos portadores y viajeros, de modo que estos últimos no pueden desplazarse a otra región sin la presencia de los primeros.

- Que afectan a la estructura de los objetos, describiéndolos como compuestos por elementos atómicos, ejemplos son los sistemas de membranas de reescritura/recombinación/contexto/inserción-eliminación (rewriting/splicing/ contextual/insertion-deletion membrane systems [Păun, 2002]).

- Sistemas P de conformación (*conformon P systems* [Frisco, 2008]), basados en las conformaciones de las estructuras moleculares que forman compuestos.

- Sistemas de membranas móviles con objetos de superficie (*mutual mobile membrane systems with surface objects* [Aman and Ciobanu, 2008, 2009]), que incorporan la abstracción de los procesos de endocitosis, exocitosis y fagocitosis.

- Sistemas P basados en energía (*energy based P systems* [Mauri et al., 2009]), que introducen el concepto de energía.

- Sistemas P estocásticos/probabilistas (*probabilistic/stochastic P systems* [Păun et al., 2010]), que introducen la probabilidad/azar en la ejecución de las reglas.

Los sistemas iniciales consideraban a la célula como un sistema completo, atendiendo únicamente a sus membranas, regiones, reglas y objetos. Su entorno era utilizado como un método de eliminación de objetos o, en algunos de ellos, como salida de la computación. No obstante, la interacción con el entorno y con otras células es un proceso esencial para la vida. De ese modo, se desarrollan nuevas variedades basadas en redes de membranas: los tejidos de sistemas P (*tissue P systems* [Carlos Martín-Vide et al., 2005]) y los sistemas P de impulsos neuronales (*spiking neural P systems* [Păun, 2008; Frisco, 2008]). Los primeros nacen como la abstracción de los tejidos biológicos, contemplando la intercomunicación celular. Subvariedades de estos son las colonias P (*P colonies* [Kelemen et al., 2004]) y las poblaciones de sistemas P (*population P systems* [Bernardini and Gheorghe, 2004]), diferenciadas en que en las últimas las células se definen como membranas elementales que se comunican entre sí a través del entorno, además de por canales de comunicación específicos y dinámicos. Por su parte, los sistemas P de impulsos neuronales [Păun, 2008; Frisco, 2008] están inspirados en las neuronas que componen el sistema nervioso. Sus principales particularidades son el hecho de que todos los nodos de la red sean membranas simples, la existencia de un único tipo de objeto que modela los impulsos nerviosos y la asignación de estados, *active* e *idle/closed*, a las membranas, que modela la limitación biológica de recibir dos impulsos consecutivos.

Actualmente, dado el elevado número de sistemas P existentes, se ha originado una rama de la disciplina que persigue la abstracción de los distintos modelos, con el ánimo de simplificar el número de variedades existentes. Fruto de este esfuerzo son las redes de células que se describirán en la próxima sección o los *kernel P systems* [Gheorghe et al., 2012].

2.1.1. Definición de un sistema de membranas

Tal y como se comentó anteriormente, existe una gran cantidad de sistemas P que se originan a partir de la definición original [Păun, 2000]. Actualmente, la tendencia es la de generar nuevos sistemas abstractos que reduzcan el número de modelos en la práctica. Así, el trabajo que se presenta en este documento está fundamentado en el entorno formal introducido en [Freund and Verlan, 2007], en el que se diseña un sistema de reescritura de multiconjuntos de clase genérica, que incluyen tanto a los sistemas P originales como a los estáticos basados en tejidos. Es por ello que en este capítulo se describen los aspectos más relevantes del citado trabajo. Para más detalles acerca de los sistemas P, se sugiere la lectura de la siguiente bibliografía: [Păun et al., 2010] y [Păun, 2002]. Para información más específica acerca del entorno formal mencionado sugerimos el trabajo citado [Freund and Verlan, 2007].

En primer lugar se presentarán algunos conceptos previos. Un alfabeto, V, es un conjunto finito o infinito de elementos denominados símbolos, o. Estos pueden combinarse para formar cadenas o palabras, w, de una determinada longitud, $|w|$, definiéndose como el número de elementos que forman la cadena. La palabra de longitud 0 se denota por λ. Del mismo modo, dada una cadena w, se emplea la notación o^k o ko con el fin de indicar el número de símbolos o contenidos en la cadena w, esto es, existen k objetos de tipo o en w. Por último, se define V^n como el conjunto de todas las cadenas de longitud n y V^* como el conjunto de todas las cadenas que pueden generarse usando los símbolos del alfabeto V.

El entorno basado en una red de células presentado en [Freund and Verlan, 2007] describe un sistema P, Π, del siguiente modo:

Definición 1. *Un sistema P, Π, se define como una red de células de grado n,*

$$\Pi = (n, V, w, Inf, R), \tag{2.1}$$

donde:

n *es el número de células (membranas), o grado del sistema P.*

V *es el alfabeto de símbolos ($o \in V$, para cada objeto presente en el sistema).*

w *es una tupla $(w_1, \ldots, w_n)$, donde w_i corresponde al multiconjunto de objetos, $o \in V$, contenido en la célula i. También puede ser denominada como configuración inicial del sistema P, y denotada como C_0.*

Inf *representa qué símbolos son suministrados por el entorno. En este caso, una célula puede recibir una cantidad infinita de objetos. De ese modo, $Inf = (Inf_1, \ldots, Inf_n)$, donde $Inf_i \subseteq V$, determina qué elementos del alfabeto son recibidos por la célula i desde el entorno.*

R *es un conjunto finito de reglas, r_i, con la forma*

$$(r_i : X \rightarrow Y; P, Q)$$

donde $X \rightarrow Y$ son vectores de multiconjuntos, $X = (x_1, \ldots, x_n)$, $Y = (y_1, \ldots, y_n)$, $x_i, y_i \in \langle V, \mathbb{N} \rangle$, $1 \leq i \leq n$, que pueden reescribirse como $((x_1, 1), \ldots, (x_n, n)) \rightarrow ((y_1, 1), \ldots, (y_n, n))$, indicando que los conjuntos x_i/y_i, ambos sobre V, son consumidos/añadidos, respectivamente, en la célula i, $1 \leq i \leq n$. Semánticamente, en el momento de aplicación de la regla, el multiconjunto X, denominado también $lhs(r_i)$ o parte izquierda de la regla (left-hand side), es consumido (deben existir suficientes recursos para que sea posible), mientras que el multiconjunto Y, denominado también $rhs(r_i)$ o parte derecha (right-hand side), es producido o generado. Atendiendo a P y Q, son vectores de multiconjuntos, denominados condiciones de permiso y condiciones de prohibición, respectivamente, pero de la forma $P = (p_1, \ldots, p_n)$ y $Q = (q_1, \ldots, q_n)$, con $p_i, q_i \in V$. Los multiconjuntos P y Q afectan a la aplicabilidad de la regla de la siguiente forma: una regla

puede aplicarse (o es aplicable) si todos los elementos de P están contenidos en las células y ninguno de los elementos de Q están contenidos en ellas. Obviamente, es necesario que también existan todos los elementos que van a ser consumidos por la regla (es decir, el multiconjunto X).

Tomando como punto de partida la definición anterior, podemos definir, a su vez, el concepto de configuración y regla aplicable:

Definición 2. *Dado un sistema P, $\Pi = (n, V, w, Inf, R)$, una configuración, C de Π es un multiconjunto $C = \{u_i\}$ sobre V con $(u_1, \ldots, u_n) \in \langle V, \mathbb{N} \rangle$, $1 \leq i \leq n$, indicando que la célula i contiene el multiconjunto de objetos u_i.*

Informalmente, se puede definir una configuración de Π como el multiconjunto de objetos presentes en Π en un instante determinado, esto es, los recursos existentes en el sistema. La configuración inicial se denota como C_0.

Definición 3. *Dado un sistema P, Π, y una configuración j-ésima tal que $C_j = \{u_1, \ldots, u_n\}$, se dice que una regla $r = (X \rightarrow Y; P, Q)$ es aplicable si, y solo si, para todo i, tal que $1 \leq i \leq n$, se cumple:*

- *Para todo $p \in p_i$, $p \subseteq u_i$. Esto es, todos los objetos de P se encuentran en la configuración C_j.*
- *Para todo $q \in q_i$, $q \nsubseteq u_i$. Esto es, ningún objeto de Q se encuentra en la configuración C_j.*
- *$x_i \subseteq u_i$. Existen suficientes recursos para que puedan ser consumidos por la regla.*

La aplicación de r sobre C_j conlleva el siguiente resultado:

- *Los objetos $u_i \in X$ son consumidos, esto es, sustraídos de la configuración C_j, dando lugar a la configuración intermedia C'_j.*
- *Los objetos $u_i \in Y$ son añadidos a la configuración intermedia C'_j, dando lugar a la siguiente configuración, C_{j+1}.*

Algoritmo 2.1 Algoritmo de Marcado. Se describe con el objetivo de introducir el concepto de multiconjunto de reglas aplicables

Requiere: Π // Sistema P
Requiere: $C = \{u_1, \ldots, u_n\}$ // Configuración
Requiere: $\mathcal{R} = \{r_1, \ldots, r_n\}$, con $r_i : X_i \to Y_i; P_i, Q_i$ // Multiconjunto de reglas
Devuelve: $True$ si todas las reglas $r \in \mathcal{R}$ pueden ser aplicadas simultáneamente, $False$ en caso contrario.

```
1. C' = C
2. para r_i ∈ R hacer
3.    si r_i es aplicable sobre C' entonces
4.       C' = C' − X_i
5.    sino
6.       devolver False
7.    fin si
8. fin para
9. devolver True
```

Con el propósito de definir el concepto de multiconjunto de reglas aplicables, es necesario introducir el Algoritmo de Marcado (Algoritmo 2.1), denotado como $Marcado(\Pi, \mathcal{R}, C)$, que determina si un multiconjunto de reglas es aplicable.

Definición 4. *Dado un sistema P, Π, y una configuración, $C = \{u_1, \ldots, u_n\}$, se define un multiconjunto de reglas aplicables sobre C, $\mathcal{R}$, como aquel multiconjunto tal que $Marcado(\Pi, \mathcal{R}, C) = True$ (Algoritmo 2.1). El conjunto de multiconjuntos aplicables sobre C se denota como $Appl(\Pi, C)$.*

Dado un sistema P, Π, una configuración, C, y un conjunto $Appl(\Pi, C)$, es posible aplicar determinadas restricciones a los elementos del conjunto $Appl(\Pi, C)$, obteniendo como resultado un subconjunto de $Appl(\Pi, C)$. Estas restricciones se denominan modo de derivación, denotado como δ. Se define el conjunto $Appl(\Pi, C, \delta)$ como el conjunto de todos los multiconjuntos de reglas aplicables que verifican δ. Nótese que $Appl$ se encuentra asociada a un sistema Π, una configuración C y un modo de derivación δ; así como $Appl(\Pi, C, \delta) \subseteq Appl(\Pi, C)$.

Existen diversos modos de derivación, por lo que únicamente se comentarán los más relevantes o de mayor interés para el trabajo que se presenta: paralelismo maximal, paralelismo minimal y paralelismo maximal de conjuntos. Se remite al lector a la referencia bibliográfica [Freund and Verlan, 2007] para más detalles

acerca de los mismos.

El modo de derivación paralelismo maximal, denotado como max, fue el primer modo de derivación y el más utilizado hasta la fecha. De manera informal, requiere que todo multiconjunto de reglas aplicables, R, pueda incluirse en *Appl* si y solo si no existe otra regla, no incluida en R, que pueda ser aplicada; en otras palabras, si se eliminaran del sistema todos los objetos consumidos por las reglas de R, no quedarían recursos suficientes para aplicar ninguna regla más. Formalmente, el paralelismo maximal se define como:

Definición 5. *El paralelismo maximal es aquel en el que,*

$$\begin{aligned} Appl(\Pi, C, max) = \{R \mid\ & R \in Appl(\Pi, C) \text{ y} \\ & \nexists R' \in Appl(\Pi, C) \text{ tal que } R' \supsetneqq R\} \end{aligned}$$

Atendiendo al modo de derivación paralelismo minimal, min, es necesario considerar el concepto de una partición de un multiconjunto de reglas, R, en subconjuntos disjuntos R_1 a R_h. Para cada conjunto $R' \subseteq R$, se define $||R'||$ como el número de conjuntos de reglas R_j, con $1 \leq j \leq h$, tal que $R_j \cap R' \neq \emptyset$. Informalmente, se puede definir como que todo multiconjunto R' debe contener al menos una regla de cada partición R_j, $1 \leq j \leq h$.

Definición 6. *El paralelismo minimal es aquel que,*

$$\begin{aligned} Appl(\Pi, C, min) = \{R' \mid\ & R' \in Appl(\Pi, C) \text{ y} \\ & \nexists R'' \in Appl(\Pi, C) \text{ tal que } R'' \supsetneqq R', \\ & (R'' - R) \cap R_j \neq \emptyset \text{ y } R' \cap R_j = \emptyset \\ & \text{para cualquier } j, 1 \leq j \leq h\} \end{aligned}$$

Por último, el modo de derivación paralelismo maximal de conjuntos, denotado como $smax$, corresponde a la ejecución maximal paralela de las reglas, pero en la que estas no pueden aplicarse más de una vez.

Definición 7. *El paralelismo maximal de conjuntos puede definirse formalmente*

como sigue:

$$S_1 = \{R \in Appl(\Pi, C) \mid |R|_{r_j} \leq 1, 1 \leq j \leq |R|\},$$
$$Appl(\Pi, C, smax) = \{R \in S_1 \mid \textit{no existe } R' \in S_1 \textit{ tal que } R' \supset R\},$$

donde $|R|_{r_j}$ *indica la multiplicidad de la regla* r_j *en el conjunto* R*, esto es, el número de veces que aparece.*

Nótese que el modo $smax$ se corresponde con el modo min_1 con una partición específica de reglas: el tamaño de la partición es $|R|$ y cada partición p_j contiene exactamente una regla $r_j \in R$.

El modo de derivación determina en gran medida cómo evoluciona el sistema desde una configuración actual, C_i, a la siguiente, C_{i+1}. Esta evolución se denomina paso de computación o transición.

Definición 8. *Dada una configuración* C_i *de un sistema P, notado como* $\Pi = (n, V, w, Inf, R)$*, con un modo de derivación* δ*, una transición se divide en tres pasos. El primero consiste en calcular el conjunto* $Appl(\Pi, C_i, \delta)$*.*

En el segundo paso, se debe elegir un elemento, $\mathcal{R}$*, de este conjunto, según la semántica del sistema. Dado que se están considerando sistemas P no deterministas,* $\mathcal{R}$ *será seleccionado empleando un método no determinista, como por ejemplo una distribución uniforme.*

Por último, la transición finaliza aplicando el multiconjunto de reglas aplicables seleccionado sobre la configuración actual, con lo que se obtiene la próxima configuración, denotado como $C_i \Longrightarrow_{(\Pi,\delta)} C_{i+1}$*.*

Informalmente, la aplicación de un paso de computación marca la temporización del modelo, esto es, el paso de computación de un sistema P se correspondería con un ciclo de reloj en un sistema *hardware* síncrono.

Atendiendo a la semántica del modelo, la computación de un sistema P comienza en la configuración inicial, C_0, a partir de la cual el sistema P evoluciona aplicando transiciones de forma sucesiva, hasta alcanzar una configuración de parada (que verifique una condición de parada). Existen varias condiciones de parada, por lo que nuevamente se remite al lector a la bibliografía [Freund and Verlan,

2007; Păun et al., 2010] y se describen, de un modo informal, las más destacadas (para ello consideremos un sistema P, Π, con un modo de derivación δ):

Condición de parada total En esta condición, el sistema P alcanza una configuración C para la que $Appl(\Pi, C, \delta) = \emptyset$. Por lo tanto, el sistema converge hacia una configuración de parada total, al no existir los recursos suficientes para que se pueda aplicar un multiconjunto de reglas aplicables que verifiquen δ.

Condición de parada adulta Esta condición considera que la ejecución del modelo Π ha finalizado cuando, al alcanzar una configuración C_i, para cualquier conjunto $\mathcal{R} \in Appl(\Pi, C_i, \delta)$, se origina la transición $C_i \Longrightarrow_{(\Pi,\delta)} C_{i+1}$, donde $C_i = C_{i+1}$. Por lo tanto, el sistema converge a una configuración de parada total adulta donde, independientemente del multiconjunto de reglas aplicables que se seleccione para ser aplicado, la configuración resultante no varía.

Desde un punto de vista práctico, puede contemplarse una condición de parada adicional que, aunque no se encuentre en la definición formal, suele ser utilizada en las implementaciones existentes. Se considera que la ejecución del sistema Π ha finalizado bajo la condición de parada límite de transiciones tras la aplicación de un número fijo de transiciones. Nótese que se trata de una condición de parada artificial, e independiente del resultado del sistema P. Generalmente es usada en problemas de simulación, donde existe una correspondencia entre el tiempo o estados del problema simulado y las transiciones del sistema, por ejemplo, en simulaciones de procesos biológicos.

Una vez presentados los sistemas P, en el siguiente apartado se describe un ejemplo que permite al lector afianzar los conceptos relativos a este modelo de computación.

2.1.2. Modelo de computación con membranas. Un ejemplo

A continuación, se muestra un ejemplo de un sistema P estático que genera el conjunto de cuadrados $\{n^2 \mid n \geq 1\}$ (Fig. 2.3). Se trata de una versión modificada por el doctorando del ejemplo del artículo fundacional [Păun, 2000]. En esta nueva versión el ejemplo original se ha adaptado al entorno de redes de células presentado

en el apartado anterior, y que será usado como referencia a lo largo de todo el documento.

En primer lugar se describe formalmente el sistema:

$$\begin{aligned}
\Pi_1 &= (3, \{a_2, b_1, c_1, d_2, e_3, f_1, g_1\}, \{\emptyset, \{a_2, d_2\}, \emptyset\}, \emptyset, \\
&\quad \{r_{1,1}, r_{1,2}, r_{1,3}, r_{1,4}, r_{2,1}, r_{2,2}, r_{2,3}\}), \\
\text{donde} & \\
r_{1,1} &= (a_2 \to a_2 b_1; \emptyset, \emptyset) \\
r_{1,2} &= (a_2 \to b_1 g_1; \emptyset, \emptyset) \\
r_{1,3} &= (d_2 \to d_2^2; \{a_2\}, \emptyset) \\
r_{1,4} &= (d_2 \to f_1; \emptyset, \{a_2\}) \\
r_{2,1} &= (b_1 \to c_1; \{g_1\}, \emptyset) \\
r_{2,2} &= (c_1 \to c_1 e_3; \{f_1^2\}, \emptyset) \\
r_{2,3} &= (f_1^2 \to f_1; \emptyset, \emptyset)
\end{aligned}$$

Este sistema cuenta con un modo de derivación paralelo maximal, max, y sus reglas son elegidas de forma no determinista del conjunto $Appl(\Pi, C_i, max)$, siguiendo una distribución uniforme discreta.

Con el propósito de ilustrar un ejemplo de computación se considera que el sistema P evoluciona tal y como se muestra en la Fig. 2.3. Según la descripción formal, los símbolos contemplados son a_2, b_1, c_1, d_2, e_3, f_1 y g_1. Para facilitar la lectura, se han añadido subíndices a los objetos y reglas que indican a qué células hacen referencia. Del mismo modo, los cambios se han resaltado en color rojo. Así, se parte de una red de tres células o membranas, de las que la célula 2 es la única que contiene objetos, concretamente el multiconjunto $\{a_2, d_2\}$.

En el primer paso de computación (Trans. 1), se tiene que el conjunto $Appl(\Pi, C_0, max) = \{\{r_{1,1}, r_{1,3}\}, \{r_{1,2}, r_{1,3}\}\}$, ya que ninguna de las reglas $r_{2,i}$ se pueden aplicar al no existir los recursos necesarios. Nótese que, aunque existen suficientes recursos para que la regla $r_{1,4}$ pueda ser aplicada, su multiconjunto de

condición de prohibición, $Q = \{a_2\}$, lo impide. La elección del conjunto de reglas aplicables es no determinista, tal y como se especificó en la definición del sistema. Para este ejemplo se considera que se aplica el primer conjunto: $\{r_{1,1}, r_{1,3}\}$. Como resultado de la computación, se dobla el número de objetos d_2, se mantiene el número de objetos $a2$ y se produce un objeto b_1.

Para las siguientes transiciones, Trans. 2, 3 y 4, que dan lugar a las configuraciones C_2, C_3 y C_4, respectivamente, se considera la misma suposición de la Trans. 1 a la hora de escoger el multiconjunto de reglas aplicables. Únicamente son aplicables las reglas $r_{1,1}$, $r_{1,2}$ y $r_{1,3}$, originando $Appl(\Pi, C_{i-1}, max) = \{\{r_{1,1}, r_{1,3}^{2^{i-1}}\}, \{r_{1,2}, r_{1,3}^{2^{i-1}}\}\}$, $2 \leq i \leq 4$. Se considera que en todas las transiciones se elige aplicar el multiconjunto $\{\{r_{1,1}, r_{1,3}^{2^{i-1}}\}$. Como resultado, en cada configuración se añade al sistema un objeto b_1 y se dobla el número de objetos d_2 existentes, mientras que el objeto a_2 se mantiene sin cambio. Nótese que, aunque existen suficientes recursos para que $r_{2,1}$ pueda ser aplicada, su multiconjunto de condición de permiso, $P = \{g_1\}$, lo impide.

Al llegar a la configuración C_4, se mantiene la situación anterior. Sin embargo, para la transición 5 se elegirá el multiconjunto $\{r_{1,2}, r_{1,3}^{16}\}\}$. Con ello, se elimina del sistema el objeto a_2, se añaden los objetos b_1 y g_1, y se dobla el número de objetos d_2, dando lugar a C_5.

Los multiconjuntos de reglas aplicables varían para la transición 6. En este caso $Appl(\Pi, C_5, max) = \{\{r_{1,4}^{32}, r_{2,1}^{5}\}\}$, por lo que la elección es sencilla. No existen recursos suficientes para que ninguna de las otras reglas puedan ser aplicadas. Por otro lado, debido a que en la transición anterior se generó un objeto g_1, ahora sí es posible aplicar $r_{2,1}$. Como resultado se obtiene la configuración C_6, en la que los objetos d_2 son sustituidos por f_1, y los b_1 por c_1.

Para las siguientes cinco transiciones, desde la 7 hasta la 11, ambas inclusive, el conjunto $Appl(\Pi, C_{i-1}, max) = \{\{r_{2,2}^{5}, r_{2,3}^{2^{11-i}}\}\}$, $7 \leq i \leq 11$, ya que no existen recursos para que se puedan aplicar el resto de reglas. En cada configuración generada, se incrementa en cinco unidades la multiplicidad del objeto e_3, al tiempo que se reducen a la mitad los objetos f_1, mientras que g_1 y c_1 permanecen constantes, dando lugar a las configuraciones C_7, C_8, C_9, C_{10} y C_{11}.

Al alcanzar la configuración C_{11}, el número de objetos f_1 es igual que la unidad. Por lo tanto, no son suficientes para que la regla $r_{2,3}$ pueda ser aplicada.

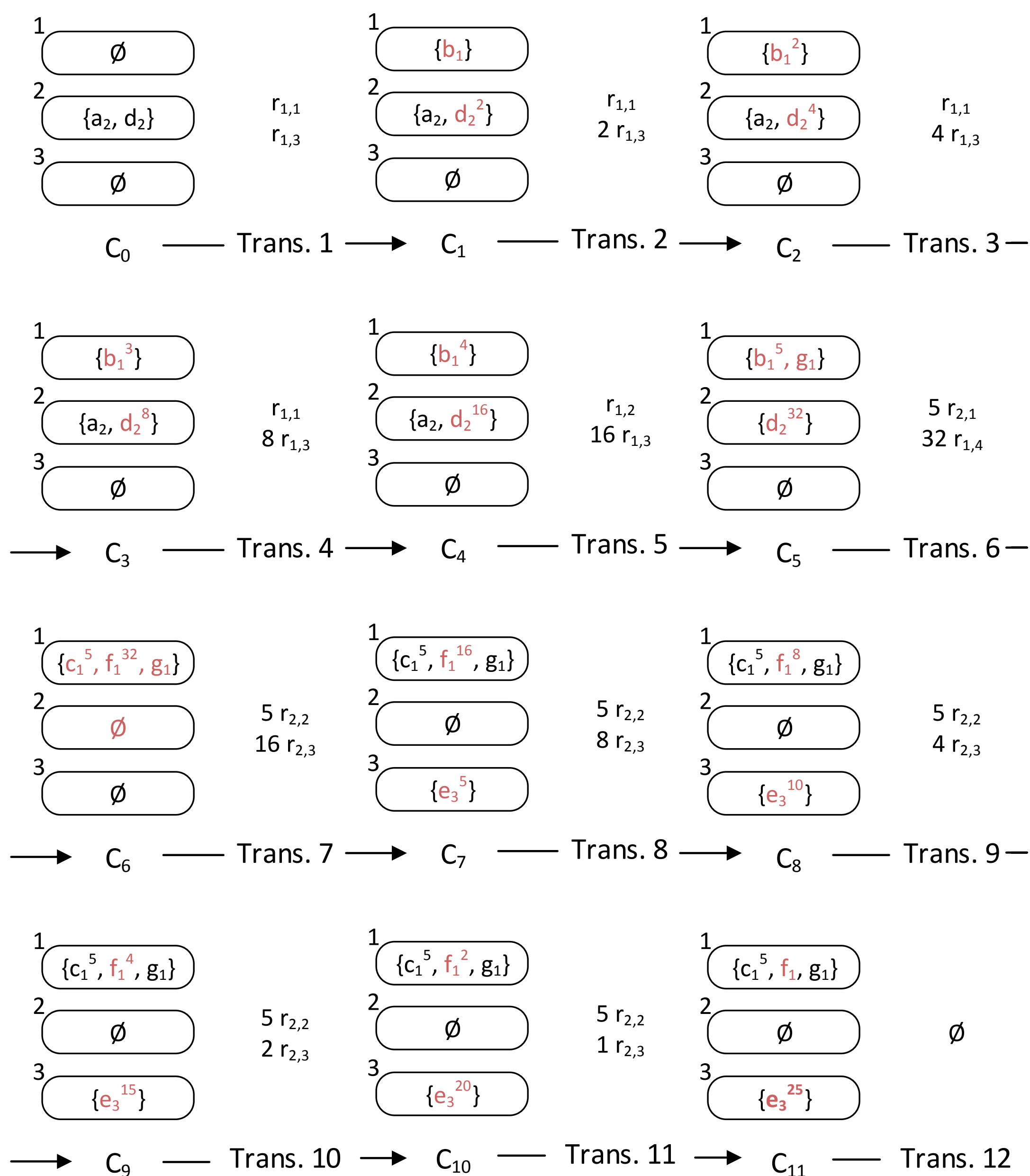


Fig. 2.3: Representación gráfica de la ejecución del sistema P de ejemplo. Se usa la notación $x\ r_{i,j}$, que es equivalente a $r_{i,j}^x$, y que representa que la regla $r_{i,j}$ es ejecutada x veces.

Por otro lado, tampoco se verifica la condición de permiso P de la regla $r_{2,2}$ ($P = \{f^2\}$). En consecuencia, ninguna regla puede ser aplicada, obteniéndose que $Appl(\Pi, C_{11}, max) = \emptyset$. Es por ello que en esta computación, el sistema P ha alcanzado una configuración final, C_{11}, verificando una condición de parada total. El cuadrado resultante es la multiplicidad de objetos e_3 presentes en el sistema, por lo que $|c_1|^2 = |e_3|$: 25 es el cuadrado de 5.

Nótese que, aunque los sistemas P sean no deterministas, desde el punto de vista del diseño es habitual emplear un no determinismo controlado, con el objetivo de simplificar las tareas y reducir el proceso de abstracción necesario, intentando que estos sean las más parecidos posibles a los requeridos para el diseño de los sistemas deterministas. En este caso, el no determinismo únicamente aparece de forma práctica en las transiciones 1-5, ya que en las siguientes $|Appl(\Pi, C_i, max)| \in \{0, 1\}$.

Como último punto, es preciso notar que el ejemplo aquí mostrado se trata de una simple demostración del funcionamiento teórico de este modelo de computación. Tal y como se describe en la siguiente sección, existen multitud de problemas en los que los sistemas P resultan de gran utilidad, siendo los más estudiados actualmente aquellos relacionados con la simulación de procesos biológicos.

2.1.3. Aplicaciones de los sistemas P

Tras los primeros desarrollos teóricos de la disciplina, los primeros trabajos de aplicación no se han hecho esperar. Esta línea se encuentra actualmente en desarrollo, por lo que no se han explorado todos los posibles campos y cuenta con un gran dinamismo. Actualmente, los ámbitos de aplicación se pueden clasificar en bio-aplicaciones, aplicaciones a la lingüística y aplicaciones a la computación.

2.1.3.1. Bio-Aplicaciones

Uno de los ámbitos de aplicación más destacados de los sistemas P en la actualidad es la modelización de procesos biológicos. Generalmente, la validación de una hipótesis precisa efectuar varias observaciones que suponen un considerable coste económico y temporal, y un error cometido a lo largo del proceso puede invalidarlas. Sin embargo, una vez diseñado y depurado un modelo, cada simulación

presenta un coste muy bajo, un tiempo de realización menor y permite un nivel de control mayor que los experimentos in vitro.

Las matemáticas continuas son las herramientas más expandidas para este propósito. No obstante, este tipo de sistemas presenta algunos inconvenientes. En primer lugar, en la mayoría de procesos biológicos existe un gran número de factores interrelacionados, que requieren de complejos sistemas de ecuaciones diferenciales y la aplicación de métodos numéricos para su resolución [Colomer et al., 2011c]. Además, una vez diseñados, la modificación de un factor generalmente implica un importante cambio en la estructura matemática del modelo, que obliga a rediseñarlo casi por completo [Nishida, 2006a]. Por último, existen situaciones en las que los resultados obtenidos con métodos continuos no se ajustan a la realidad [Ciobanu, 2006].

Por otro lado, los sistemas P ofrecen una serie de características muy atractivas para este ámbito [Colomer et al., 2011c; Ciobanu, 2006; Gutiérrez-Naranjo et al., 2005b]. Son modulares, por lo que la inclusión de modificaciones no altera significativamente el modelo, y su complejidad es inferior a la de los modelos de ecuaciones diferenciales. A diferencia de estos últimos, no poseen una limitación en el número de factores interrelacionados a considerar, y admiten diferentes enfoques: discreto, probabilista o continuo [Cordón-Franco and Sancho-Caparrini, 2005; Colomer et al., 2011b; Pérez-Jiménez and Romero-Campero, 2007], por lo que el modelo puede representar distintos aspectos de los procesos biológicos.

Los procesos modelados pueden ser macroscópicos o microscópicos. En el primer caso, existen modelizaciones de ecosistemas con el fin de mejorar la gestión de espacios protegidos y/o especies en riesgo por parte de las autoridades competentes. En esta línea se enmarcan los trabajos efectuados por grupos de investigación de las Universidades de Sevilla, Lleida, Barcelona y Autónoma de Barcelona, en colaboración con el Grupo de Estudio y Protección del Quebrantahuesos (Lleida) y el Instituto Pirenaico de Ecología del CSIC. Los ecosistemas modelados son el del quebrantahuesos (*gypaetus barbatus*) [Cardona et al., 2008, 2009, 2010; Colomer et al., 2011b; Fondevilla et al., 2011], el del mejillón cebra (*dreissena polymorpha*) [Cardona et al., 2010], el del rebeco pirenaico o sarrio (*rupicapra pyrenaica*) [Colomer et al., 2011a] y el del tritón pirenaico (*calotriton asper*) [Colomer et al., 2011c].

La simulación de procesos biomoleculares pertenece al ámbito microscópico. Los primeros trabajos están basados en sistemas P estocásticos [Andrei and Mario, 2006; Cazzaniga et al., 2006]. Algunos ejemplos son la modelización del proceso biológico de la bomba sodio-potasio [Besozzi and Ciobanu, 2005], de la predicción de tumores en la etapa prevascular [Gutiérrez-Naranjo et al., 2005b], de los canales mecanosensitivos de gran conductancia [Ardelean et al., 2006] y la respiración [Cavaliere and Ardelean, 2006] en la bacteria E. Coli. La mayoría de los resultados a partir del año 2006 usan como base los sistemas P metabólicos [Manca, 2008, 2010] específicamente diseñados para este fin, como la modelización de las redes de transducción de señales biológicas [Castellini and Franco, 2008], del fenómeno *Non Photochemical Quenching* (NPQ) [Manca et al., 2009], que es un procedimiento usado por las plantas en condiciones de exceso de luz, y de una red de regulación genética para el análisis de la expresión génica [Marchetti and Manca, 2011].

2.1.3.2. Aplicaciones a la computación

La resolución de problemas NP en un tiempo aceptable (aunque este término no se definirá, se preferirá que este sea de complejidad menor o igual a polinomial) es uno de los grandes retos a los que se enfrenta la computación actual. Este interés se justifica en el hecho de que existe gran cantidad de este tipo de problemas con aplicación práctica. Es por ello que si se demuestra que $P \neq NP$ (y mientras no se demuestre lo contrario) será necesario estudiar métodos de aproximación. En este sentido, el paralelismo maximal, el ser computacionalmente universal y la *habilidad* de crear un espacio exponencial en un tiempo polinomial mediante la división de membranas sitúan a los sistemas P como un modelo de computación idóneo para este fin [Pérez-Jiménez, 2005; Pérez-Jiménez et al., 2006; Nishida, 2006b], consiguiendo resolver problemas NP completos en tiempo polinomial e incluso lineal [Păun, 2001] y constante [Leporati et al., 2007] (con una configuración *semi-uniform*[1]). De ese modo, se han diseñado sistemas P que resuelven diversos problemas de índole matemática y que tienen gran importancia en el campo de las Ciencias de la Computación [Alhazov and Sburlan, 2006; Leporati et al., 2007], incluso aplicando optimizaciones con ideas de otros ámbitos, como el uso de estra-

[1]Una configuración *semi-uniform* implica un procesamiento previo no contemplado en el orden de complejidad para cada instancia del problema.

tegias de búsqueda procedentes de la inteligencia artificial [Gutiérrez-Naranjo and Pérez-Jiménez, 2011].

El procesamiento de imágenes es un subcampo de aplicación que posee gran proyección debido al amplio elenco de materias donde puede aplicarse. En [Daniel et al., 2010] se presenta un sistema P encuadrado en el marco de los sistemas P catalíticos con el fin de calcular la homología a imágenes $2D$. Aunque este subámbito es reciente, algunos autores han presentado variantes como los tejidos de sistemas P con gramáticas de array [Christinal et al., 2010], utilizados para el cálculo de la segmentación de imágenes en $2D$.

Los sistemas multiagentes se caracterizan por ser interactivos debido a los procesos de comunicación entre agentes; paralelos ya que varios agentes actúan al mismo tiempo; y dinámicos, pudiendo producirse cambios en su organización, roles de agentes, configuración, etc. Es por ello que la computación por membranas se presenta como una posible alternativa para su modelización [Stamatopoulou et al., 2005] con cualquier tipo de agentes [Kefalas and Stamatopoulou, 2011] (reactivos, proactivos e híbridos).

Por último, la aplicación en redes de ordenadores es también motivo de estudio. La modelización y análisis de redes con políticas de seguridad basadas en firewall [Leporati and Ferretti, 2010a,b], la creación de protocolos para autenticación de mensajes [Atanasiu, 2003] o la implementación de algoritmos de ranking para páginas web [Muskulus, 2009] son algunos ejemplos.

2.1.3.3. Aplicaciones a la lingüística

La lingüística constituye otro campo de aplicación. Sus posibles usos [Enguix, 2004; Enguix and Jiménez López, 2006; Gramatovici and Enguix, 2006; Alhazov et al., 2009] son el estudio de la semántica, del desarrollo del lenguaje, de la sociolingüística, del diálogo, de la anáfora, de la flexión lingüística y el procesamiento de lenguaje natural. En [Enguix and Jiménez López, 2006] se ha desarrollado un nuevo tipo de sistema P denominado *Linguistic P System* (LPS) con dos variantes: *Conversational P Systems* (CPS) y *Dynamic Meaning P Systems* (DMPS), aplicados al proceso de diálogo y a la semántica léxica. Sus principales características son la relación membrana-contexto, de modo que las membranas representan

contextos y estos, a su vez, lenguajes, estados del lenguaje (evolución de este), sociedades o grupos sociales (sociolingüística), contextos (semántica), etc.; la existencia de varios alfabetos, de modo que una membrana puede tener asociado uno o varios; y la posibilidad de que cualquier elemento de un LPS (alfabetos, dominios, estructuras del lenguaje, reglas e interacción entre membranas) pueda evolucionar durante una computación.

2.2. Implementaciones actuales

En esta sección se describe la evolución de las distintas implementaciones existentes para sistemas P. No obstante, antes de describir los distintos trabajos, se detallan aquellos requisitos de diseño más relevantes para este fin, y que serán empleados en la comparación de los distintos trabajos.

2.2.1. Requisitos de diseño

Todo trabajo de implementación lleva asociado un conjunto de requerimientos que en ocasiones prioriza y/o descarta el uso de determinadas arquitecturas y tecnologías. La potencia de cómputo, escalabilidad y flexibilidad son los atributos de calidad más destacados que se deben valorar en el diseño de una implementación de un sistema P. La relación entre estos factores rara vez es directamente proporcional, siendo labor del diseñador lograr el mejor equilibrio posible. A continuación, se describen brevemente cada uno de estos requisitos.

Potencia de cómputo

La potencia de cómputo se define como la cantidad de trabajo ejecutado o realizado por el sistema en una unidad de tiempo; esto es, la rapidez con la que es ejecutado un sistema P. El modelo de computación es paralelo y, a excepción de algunas extensiones, no contempla ninguna limitación física, por lo que el número de membranas, objetos y reglas a considerar y ejecutar simultáneamente no está acotado. Todo ello lo convierte en un factor crítico que determinará el tamaño del problema que es posible resolver en un tiempo adecuado.

Escalabilidad

La escalabilidad indica la capacidad del sistema para mantener estable el rendimiento ante un crecimiento continuo de trabajo. Su consideración se encuentra justificada, al igual que en el atributo anterior, por la ausencia de límite en el número de elementos que lo componen y que pueden crearse durante una computación en un sistema P. Es un elemento crítico si se desea dotar de utilidad práctica a la plataforma, especialmente en lo que respecta a la creación de objetos, ejecución de reglas y creación de membranas en sistemas con membranas activas.

Flexibilidad

La computación con membranas es un modelo de computación orientado a máquinas; en consecuencia, para resolver una instancia de un problema se requiere un sistema P específico. Por lo tanto, la arquitectura debe ser lo suficientemente flexible como para permitir la adaptación de un diseño general al mayor número de instancias posibles de un problema, así como al mayor número de estos.

Por otro lado, este parámetro debe también considerarse desde el punto de vista de añadir nueva funcionalidad al sistema, esto es, lo que se traduce en implementar nuevas variedades de sistemas P.

2.2.2. Trabajos de implementación *software*

Desde el nacimiento de la disciplina, la gran mayoría de los trabajos de implementación emplean tecnología *software*, empezando por implementaciones secuenciales, para después centrarse en el desarrollo de soluciones paralelas y distribuidas. En la actualidad, la práctica totalidad de los esfuerzos se encuentran centrados en el desarrollo sobre tecnología GPU.

2.2.2.1. Trabajos de implementación con tecnología *software* secuencial

Los primeros trabajos aparecen en el año 2000, dos años después del artículo fundacional, consistiendo en implementaciones *software* secuenciales. En [Maliţa,

2000] M. Maliţa presenta una implementación de sistemas P transitivos empleando el lenguaje de programación LPA-Prolog, en el que la autora persigue la transparencia en el desarrollo, primando la comprensión frente a la eficiencia. En el mismo año Y. Suzuki y H. Tanaka presentan en [Suzuki and Tanaka, 2000] una implementación en LISP de un sistema P transitivo limitando el tamaño de los multiconjuntos y para una estructura de membranas estática, eliminando las reglas de disolución, obteniendo una variación denominada *Artificial Cell Systems* (ACS).

El Grupo de Computación Natural de la Universidad Politécnica de Madrid (UPM) ha iniciado una línea de investigación en este campo, presentando los primeros trabajos en el año 2001. Estos se centran en cuestiones de algoritmia y de implementación [Arroyo et al., 2001a,b; Baranda et al., 2001], publicando los primeros sistemas completos en los dos años posteriores, en los que se presenta una implementación *software* secuencial de los sistemas P transitivos empleando el lenguaje de programación Haskell [Baranda et al., 2002; Arroyo et al., 2003].

Otra línea de trabajo es constituida por el Grupo de Computación Natural de la Universidad de Sevilla (RGNC). Los primeros trabajos se centran en los sistemas P transitivos, presentando implementaciones secuenciales utilizando MzScheme [Balbotín-Noval et al., 2003] y el *software* SimCM [Nepomuceno-Chamorro, 2004], implementado en Java con soporte multihilo y que constituye una herramienta gráfica para el desarrollo de sistemas. Además, en [Nepomuceno et al., 2005] se presenta una herramienta que permite describir sistemas empleando el formato *Systems Biology Markup Language* (SBML)[2], siendo necesaria la conversión a CLIPS (implementada por el *software*) para la ejecución. En trabajos posteriores se desarrollan implementaciones secuenciales de sistemas P con membranas activas utilizando Prolog [Cordón-Franco et al., 2004; Gutiérrez-Naranjo et al., 2005a] y CLIPS [Pérez-Jiménez and Romero-Campero, 2004]. Los autores no se limitan a la implementación de sistemas, sino que desarrollan herramientas de desarrollo y depuración, que culminan con el desarrollo de la plataforma P-Lingua[3] [Díaz-Pernil et al., 2008, 2009]. Esta engloba un lenguaje específico de programación para sistemas P, un compilador y una implementación con soporte multihilo desarrollado en Java, además de presentar un diseño modular que admite el uso de

[2]SBML: `http://sbml.org/index.psp`

[3]Para más información sobre P-Lingua consulte `http://www.p-lingua.org/`.

otras implementaciones. P-Lingua ha sido utilizada en diferentes trabajos de aplicación llevados a cabo por este grupo [Pérez-Hurtado et al., 2010; Martínez-del Amor et al., 2010]; actualmente se encuentra en la versión 2.0 [García-Quismondo et al., 2010] y recientemente se ha desarrollado una nueva implementación para dar soporte a los sistemas P de impulsos neuronales [Macías-Ramos et al., 2011].

Por su parte, el grupo de Modelos de Computación Natural de la Universidad de Verona ha centrado sus esfuerzos en el desarrollo de implementaciones de sistemas P metabólicos. El resultado es el *software* MPsim [Bianco et al., 2006; Bianco and Castellini, 2007], desarrollado en JAVA con soporte multihilo basado en una arquitectura de *plugins* que le confiere modularidad y fácil extensibilidad. A partir de este *software* se desarrolló MetaPlab [Castellini and Manca, 2009], que amplía la funcionalidad del anterior y que está orientado también a la simulación de procesos biológicos.

Las herramientas *software* más maduras presentan la ventaja característica de este tipo de implementaciones: flexibilidad, debido a su modularidad y extensibilidad. No obstante, el tamaño de los problemas que pueden resolverse con estas es limitado, ofreciendo una limitada escalabilidad y potencia de cómputo. Es por ello que en algunas se imponen restricciones a los sistemas o su uso se reduce a problemas no suficientemente grandes. Para problemas más exigentes es necesario emplear otros sistemas con mayor potencia de cómputo. Sin embargo, la modularidad de las plataformas desarrolladas permite reutilizar los módulos de depuración, interfaz y compilación, añadiendo las nuevas implementaciones como módulos o *plugins*.

2.2.2.2. Trabajos de implementación con tecnología CUDA

La inclusión de la arquitectura Tesla en las GPU y el desarrollo de CUDA ha permitido ofrecer potentes unidades de multiprocesamiento de bajo coste, y en consecuencia los primeros resultados no se han hecho esperar.

La línea de investigación más activa es la constituida por miembros del RGNC y del Grupo de Arquitectura y Computación Paralela de la Universidad de Murcia. Para su diseño se divide la ejecución de un sistema P en dos fases ejecutadas secuencialmente: selección de reglas, donde se determina qué reglas se usarán; y ejecución,

donde se procede a la ejecución de las reglas previamente seleccionadas. En base a esta división, se han presentado dos implementaciones: *(1)* híbrida [Martínez-del Amor et al., 2009], en la que la fase de selección se efectúa en la GPU y la de ejecución en la *Central Processing Unit* (CPU) y *(2)* masivamente paralela implementada (ambas) por completo en la GPU [Cecilia et al., 2009a]. La arquitectura Tesla y el diseño del algoritmo imponen dos restricciones en los sistemas aceptados por la implementación: por razones de sincronización el número máximo de niveles permitido en la jerarquía de membranas es solo 2 y el número de elementos del alfabeto debe ser divisible por un número inferior a 512. Se han efectuado pruebas comparando estas implementaciones y otra secuencial tomando como base: (1) el problema de las N-Reinas[4] [Cecilia et al., 2010a], (2) una serie de problemas generados para este fin [Guerrero et al., 2009; Cecilia et al., 2009b] y (3) el problema SAT[5] [Cecilia et al., 2010b]. Los resultados se muestran en base a dos parámetros: el número de objetos por membrana y el número de membranas. En relación a la implementación masivamente paralela, cuando el número de objetos por membrana permanece constante y se aumenta el número de membranas el tiempo es lineal hasta alcanzar el número de recursos de la GPU, a partir del cual el tiempo crece exponencialmente, aunque manteniéndose siempre dos órdenes de magnitud por debajo de la implementación secuencial. El caso opuesto se comporta de forma similar, pero manteniéndose tres órdenes de magnitud por debajo. Con respecto a la implementación híbrida, esta ofrece peores resultados debido a la penalización por transmisión por el bus PCI. Los últimos resultados están orientados a la implementación de nuevos algoritmos con el propósito de desarrollar implementaciones más eficientes de sistemas P probabilísticos que modelan procesos biológicos, concretamente los Sistemas P de Dinámica de Poblaciones (PDP) [Colomer et al., 2010]: *Binomial Block Based Algorithm* (BBB) [Cardona et al., 2010], *Direct Non-Deterministic distribution with Probabilities* (DNDP) [Martínez-del Amor et al., 2010; Martínez-del Amor et al., 2011], y *Direct distribution based on Consistent*

[4] El problema de las N-Reinas es un pasatiempo en el que se colocan n reinas en un tablero de ajedrez sin que se amenacen.

[5] El problema de satisfacibilidad booleana o SAT consiste en conocer si una expresión booleana con variables y sin cuantificadores tiene asociada una asignación de valores para sus variables que hace que la expresión sea verdadera. Un ejemplo de una instancia del problema SAT sería conocer si existen valores para x_1, x_2, x_3 y x_4 tal que la expresión $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4)$ sea cierta.

Blocks Algorithm (DCBA) [Martínez-del Amor et al., 2013].

El RGNC también colabora con miembros del Departamento de Ciencias de la Computación de la Universidad de Philippines Diliman en el desarrollo de una implementación de sistemas P de impulsos neuronales [Cabarle et al., 2011a,b,c]. El hecho de que esta variedad pueda ser representada matricialmente [Zeng et al., 2011] la hace más adecuada que otras para su implementación en GPU. Actualmente se trata de una implementación híbrida que experimentalmente ha obtenido un rendimiento superior a implementaciones secuenciales, que comprenden desde una mejora en el tiempo de cómputo de 1,4 % para sistemas pequeños a 6,8 % para sistemas de mayor tamaño. Además, ofrece una ecuación para determinar el tamaño máximo del sistema que puede ser aceptado, obteniendo que el número de reglas es el único factor que incide en este límite. Las líneas de futuro persiguen la integración con P-Lingua y la optimización de la implementación.

Las implementaciones basadas en GPU ofrecen un compendio entre potencia y esfuerzo de desarrollo, presentando una buena flexibilidad, al tratarse de un desarrollo *software*. Respecto a la escalabilidad, poseen un notable grado de paralelismo en un único dispositivo, lo que elimina gran parte de los problemas asociados a los sistemas distribuidos. Por otro lado, al tratarse de dispositivos dedicados, presentan una mayor potencia computacional que los procesados de propósito general. Su principal desventaja es su especificidad, que no las hace adecuadas para determinados algoritmos. Actualmente, su grado de desarrollo ha alcanzando un cierto grado de madurez, con implementaciones específicas.

2.2.2.3. Trabajos de implementación con sistemas paralelos y distribuidos

Los sistemas P son en esencia sistemas distribuidos con un alto grado de paralelismo, lo que conlleva contemplarlos como candidatos a la hora de realizar implementaciones distribuidas. En este sentido, destacan los trabajos realizados por G. Ciobanu [Ciobanu and Paraschiv, 2002; Ciobanu et al., 2003a,b; Ciobanu, 2003; Ciobanu and Guo, 2004] y A. Syropoulos [Syropoulos et al., 2004].

El trabajo de G. Ciobanu fue el primero en contemplar el uso de los sistemas distribuidos tras desarrollar una implementación secuencial para sistemas P con

membranas activas [Ciobanu and Paraschiv, 2002]. Además, propone una nueva variante, el *Client-Server P System* (CSPS) [Ciobanu et al., 2003b], computacionalmente universal, enfocada a la modelización de procesos moleculares y basada en los procesos de comunicación, sin reglas de creación y/o destrucción de objetos, que la hace idónea para la implementación en arquitecturas cliente-servidor. Con respecto a los sistemas distribuidos, su desarrollo se basa en la librería *Message Passing Interface* (MPI), usando el lenguaje *C++* [Ciobanu et al., 2003a]. En sus publicaciones no ofrece resultados exhaustivos, aunque describe con detalle la implementación efectuada y los algoritmos [Ciobanu, 2003] con los que hacer frente a los problemas potenciales de este tipo de sistemas: comunicación, memoria compartida, exclusión mutua y tolerancia a fallos. En [Ciobanu and Guo, 2004] describe una prueba llevada a cabo en un clúster de ordenadores, formado por 64 nodos, cada uno de ellos compuesto por dos equipos con procesadores Intel PIII a 1,4GHz y 1GB de memoria, e interconectados mediante una red Ethernet con enlaces gigabit. Aunque esta topología implementa el paralelismo a nivel de membranas, el hecho de que cada nodo esté compuesto por dos núcleos de procesado impide el segundo nivel de paralelismo, a nivel de reglas. Según el autor, los principales problemas residen en los procesos de comunicación y cooperación, hasta tal punto que el tiempo de comunicación es mayor que el de ejecución, agravado por problemas de congestión de la red detectados durante las pruebas. Es por ello que su reducción se fija como principal objetivo para posteriores esfuerzos.

Por su parte, A. Syropoulos ha implementado un sistema P transitivo tomando como base el protocolo *Remote Method Indication* (RMI) de Java [Syropoulos et al., 2004]. Cada nodo representa una membrana, eligiéndose aleatoriamente aquel que representa la membrana piel, aunque la ejecución de los hilos puede llevarse a cabo en otra máquina distinta. A pesar de que no se ofrece ningún resultado de rendimiento, la asociación membrana-nodo constituye una fuerte restricción al sistema. Además, cita que es necesario desarrollar distintas mejoras de cara a la eficiencia del mismo.

Aparte de las líneas mencionadas no se han desarrollado más trabajos de suficiente entidad en este área. Los resultados obtenidos subrayan las carencias de estas plataformas para ejecutar sistemas P: un esfuerzo de desarrollo notable para mitigar sus problemas inherentes de comunicación y sincronización y una escalabilidad

y flexibilidad reducida, ya que el número de nodos es limitado, no se implementa el doble paralelismo y la adición de nuevos nodos implica la adquisición de un nuevo equipo completo, además de aumentar los problemas de comunicación de la red de interconexión. Respecto a la potencia de cómputo, los problemas que presentan estas soluciones no suponen un avance significativo frente a la implementaciones secuenciales. En este sentido, Ciobanu establece que su reducción constituye el único medio para aumentar considerablemente las prestaciones, y la establece como principal línea de futuro.

2.2.3. Trabajos de implementación *hardware*

Los avances más significativos en este área son mérito de tres líneas de investigación: B. Petreska, autora de la primera implementación *hardware*; el Grupo de Computación natural de la UPM, destacando su trabajo en la realización de algoritmos y estructuras de datos; y V. Nguyen, diseñando dos arquitecturas y una herramienta *software* de generación automática.

2.2.3.1. La solución *hardware* de B. Petreska

La primera implementación *hardware* empleando una FPGA es fruto del trabajo llevado a cabo por B. Petreska y C. Teuscher [Petreska and Teuscher, 2004]. El sistema P ejecutado es transitivo, considerando la creación y disolución de membranas. Su diseño está basado en la interconexión de unidades funcionales de un único tipo, el *membrane hardware component*. Las comunicaciones son punto a punto, siguiendo una topología mixta con una organización que busca minimizar el número de líneas de comunicación (Fig. 2.4). La unidad funcional representa una membrana y está compuesta por un conjunto de registros y un reactor (Fig. 2.5). Los registros almacenan toda la información relativa a la membrana: la cardinalidad de los objetos presentes, la etiqueta, el estado y las reglas, junto con las relaciones de prioridad de estas. El reactor constituye el módulo de control y la interfaz de entrada y salida.

La ejecución se divide en macro-pasos que, a su vez, se componen de micro-pasos. El micro-paso se ejecuta en el contexto local de una unidad funcional, y consiste en seleccionar y aplicar la siguiente regla atendiendo a las relaciones de

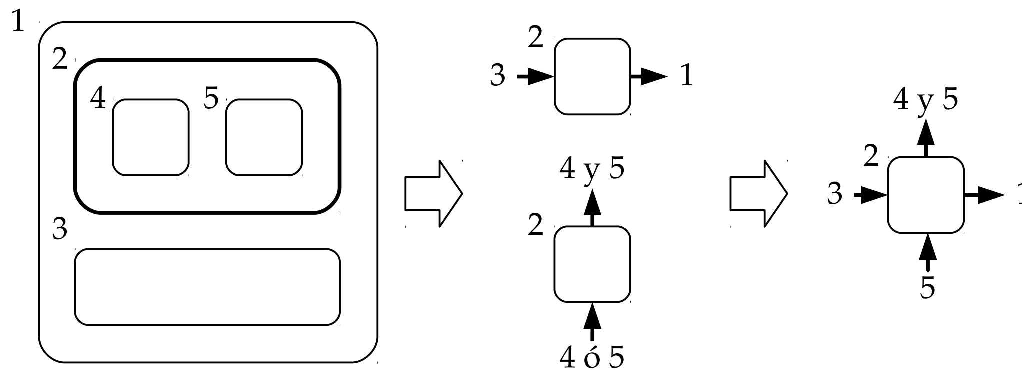


Fig. 2.4: Esquema de la interconexión de membranas empleado por Petreska en su arquitectura [Petreska and Teuscher, 2004].

prioridad y los recursos disponibles. Un macro-paso se corresponde con un paso de ejecución del sistema de membranas, esto es, a una transición de configuraciones, comprendiendo la aplicación de todos los micro-pasos posibles. La ejecución de los micro-pasos es secuencial en cada unidad funcional, aunque el conjunto de todas ellas trabaja de forma paralela.

Atendiendo a la metodología de trabajo con el sistema, esta se basa en una aplicación *Java* que genera código de descripción de *hardware* que, junto a una serie de *scripts*, permite adaptar el diseño genérico de la arquitectura al sistema de entrada. El conjunto es ejecutado, sintetizado, implementado y programado en el dispositivo con el empleo de las herramientas ModelSim VHDL Simulator y Leonardo VHDL Synthesizer de Mentor Graphics[6] y varias utilidades de Xilinx.

El trabajo de Petreska es un primer acercamiento y demuestra que la tecnología FPGA puede ser válida para la ejecución de sistemas P. La influencia del modelo teórico es patente en su diseño: unidades funcionales que representan membranas interconectadas. No obstante, posee determinados aspectos desaconsejables teniendo en cuenta las motivaciones de este trabajo: el diseño *hardware* de un sistema paralelo con la potencia, escalabilidad y flexibilidad necesarias como para ejecutar de forma eficiente y en un tiempo razonable sistemas P suficientemente grandes.

Los sistemas P poseen un doble paralelismo, a nivel de membrana y a nivel de

[6] Mentor Graphics: `https://www.mentor.com/`

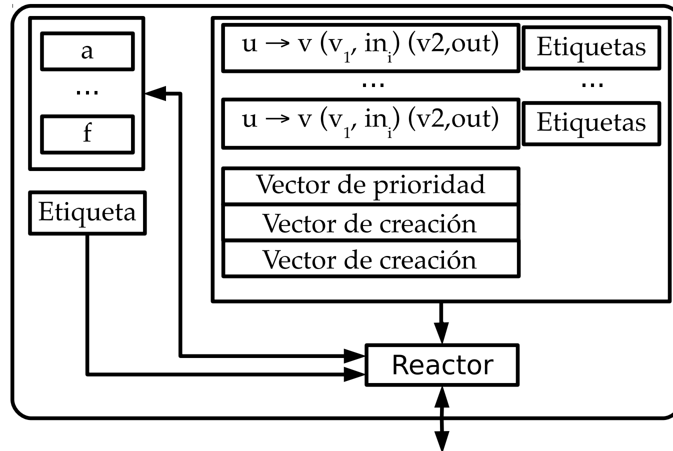


Fig. 2.5: Esquema de un *membrane hardware component*: unidad funcional básica de la arquitectura de Petreska [Petreska and Teuscher, 2004].

reglas; no así el trabajo descrito, en el que únicamente se implementa el primer nivel con la ejecución en paralelo de los *membrane hardware component*. Localmente las reglas son ejecutadas de forma secuencial, con la consiguiente pérdida de potencia de cómputo del modelo. Además, las reglas son ejecutadas de forma individual; la obtención de un número de aplicabilidad en cada macro-paso reduciría considerablemente el tiempo de ejecución.

Por otro lado, las unidades funcionales presentan una estructura estática que agrava la flexibilidad de la implementación. Algunas limitaciones son que: el número de tipos de objetos participantes en las reglas es fijo o que únicamente se permita un elemento de tipo *in* en cada una. Además, la adaptación a un sistema P es limitada, debido a que la metodología está basada en *scripts* que modifican una implementación ya realizada restando flexibilidad a la implementación.

En tercer lugar, las estructuras de datos estáticas, la replicación de módulos, y la implementación de los procesos de creación y disolución de membranas mediante bits de estado, implementando desde el inicio todas las unidades funcionales, incluso las aún no creadas, y el hecho de mantener las disueltas tienen como consecuencia un desperdicio de recursos *hardware* en la FPGA. Una de las posibles soluciones a este problema consistiría en hacer uso de una de las características distintivas de esta tecnología: la reconfiguración dinámica.

Por último, el empleo de un único tipo de unidad funcional y la rigidez de su arquitectura, unido a una topología de red muy específica dificultan la adición de características de nuevas variedades de sistemas, limitando la extensibilidad (flexibilidad) de la implementación.

2.2.3.2. Trabajo del Grupo de Computación Natural de la UPM

El Grupo de Computación Natural de la UPM ha desarrollado una línea de implementaciones de sistemas P empleando soluciones *software* y *hardware*. Con respecto a este último tipo, sus publicaciones se centran en el diseño de arquitecturas *hardware* teóricas para la ejecución de sistemas P transitivos con soporte de disolución de membranas. Su trabajo se enfoca en el uso de procesadores de membranas, de modo que cada membrana tiene asociado un procesador encargado de procesar sus reglas. Aunque la arquitectura teórica está desarrollada casi por completo, los autores no ofrecen ningún resultado de implementación.

En [Arroyo et al., 2004a] se define la red de interconexión de los procesadores, que recoge la estructura de membranas del sistema. El problema de la disolución de membranas es resuelto estableciendo estados en los procesadores: modo activo o pasivo. En el modo activo el procesador representa una membrana no disuelta, asumiendo las funciones de ejecución de reglas y comunicación con los que representan las membranas hijas y padre, en caso de que las hubiera. En modo pasivo el procesador actúa como un puente de comunicación. La implementación de estados está basada en la asociación de un vector de procesado $Id_{Proc}(n)$ y de un vector de máscara $Mask_{Proc}(n)$ por cada procesador n, dando lugar a las matrices de conectividad. A partir de estos se describen las operaciones necesarias para los procesos de comunicación y actualización de la estructura de membranas.

Otro aspecto a definir es el modo de representación de los objetos presentes en el interior de las membranas y las reglas [Arroyo et al., 2004b]. Para el primer caso se emplea un vector que contiene en cada posición la cardinalidad de un objeto, siendo necesario establecer una relación de orden entre estos. Las reglas son representadas con un vector con múltiples campos que indican el estado de la regla, si es aplicable, la etiqueta, a qué procesadores (membranas) envían objetos y un vector con los objetos que participan en la regla. Así, considerando las reglas

Algoritmo 2.2 Algoritmo paso a paso [Fernández, 2006]. Donde $\phi_{M(R(U))}$ es el conjunto vacío, $w_{R(U)}$ es el conjunto de pares (regla, número de aplicación) que define la transición a la siguiente configuración, w representa al multiconjunto de objetos en la configuración actual, A es el conjunto de reglas de evolución e $input(r_i)$ indica los objetos consumidos por la regla r_i.

1. $w_{R(U)} \leftarrow \phi_{M(R(U))}$
2. **repetir**
3. $\quad r_i \leftarrow random(A)$
4. $\quad$ **si no** $applicable(r_i, w)$ **entonces**
5. $\quad\quad A \leftarrow A - \{r_i\}$
6. $\quad$ **sino**
7. $\quad\quad w_{R(U)} = w_{R(U)} + \{(r_i, 1)\}$
8. $\quad\quad w \leftarrow w - input(r_i)$
9. $\quad$ **fin si**
10. **hasta** $|A| = 0$

de la forma $u \longrightarrow v$, el último vector posee $n+2$ posiciones: las $n+1$ primeras se corresponden con vectores con una posición por objeto, de los que el primero representa la parte izquierda de la regla y el resto la derecha, de modo que para cada membrana se enumera la cantidad de objetos que recibe al ejecutarse la regla; con respecto a la última posición, establece la disolución de la membrana. Con estas estructuras de datos, cada procesador debe mantener la cardinalidad de los objetos presentes en la membrana, las reglas y las relaciones de prioridad entre estas.

Una vez establecidas las estructuras de datos, resta indicar la selección y aplicación de reglas para alcanzar nuevas configuraciones a partir de la inicial. El primer paso consiste en seleccionar las reglas activas, que son aquellas aplicables (existen suficientes objetos para ser aplicadas), útiles (envían objetos a membranas no disueltas) y para las que no existe una regla de mayor prioridad aplicable y útil. Empleando puertas AND, puertas OR y comparadores y, en base a la estructura de datos previamente definida, el autor construye circuitos para determinar las reglas activas [Fernández et al., 2005].

El siguiente paso consiste en determinar el número de aplicabilidad para cada una de las reglas activas. En el desarrollo teórico [Fernández, 2006] se presentan tres algoritmos: aplicación paso a paso (Algoritmo 2.2), aplicación con *benchmark*

Algoritmo 2.3 Algoritmo de aplicación con *benchmark* de maximalidad [Fernández, 2006]. Donde $\phi_{M(R(U))}$ es el conjunto vacío, $w_{R(U)}$ es el conjunto de pares (regla, número de aplicación) que define la transición a la siguiente configuración, w representa al multiconjunto de objetos en la configuración actual, A es el conjunto de reglas de evolución e $input(r_i)$ indica los objetos consumidos por la regla r_i.

1. $w_{R(U)} \leftarrow \phi_{M(R(U))}$
2. **repetir**
3. $r_i \leftarrow random(A)$
4. $max \leftarrow maximalApplicable(r_i, w)$
5. **si** $max = 0$ **entonces**
6. $A \leftarrow A - \{r_i\}$
7. **sino**
8. $k \leftarrow random(1, max)$
9. $w_{R(U)} = w_{R(U)} + \{(r_i, k)\}$
10. $w \leftarrow w - k * input(r_i)$
11. **fin si**
12. **hasta** $|A| = 0$

de maximalidad (Algoritmo 2.3) y aplicación con *benchmark* de minimalidad (Algoritmo 2.4). El primero consiste en un bucle en el que en cada paso se selecciona una regla a la que se incrementa en 1 su número de aplicabilidad. Con respecto a los otro dos, en cada paso se selecciona aleatoriamente el número de veces que se aplicará una regla aplicable: en el de maximalidad el número está comprendido en el intervalo $[1, N_{ap}(r, C'_i)]$, mientras que en el de minimalidad pertenece al intervalo $[min, N_{ap}(r, C_i)]$, donde $min = N_{ap}(r, C_i - C'_i)$, C_i se corresponde con la configuración i-ésima y C'_i con la configuración parcial obtenida al eliminar en cada iteración los objetos correspondientes de C_i (Algoritmo 2.3). Nótese que no se considera que una regla aplicable pueda no ser elegida para ser ejecutada, ya que tanto para el benchmark de maximalidad como de minimalidad el menor valor posible es 1. En los resultados obtenidos se muestra una reducción de tiempo de hasta el 84,7 % para el *benchmark* de maximalidad y del 70,2 % para el *benchmark* de minimalidad, ambos con un orden logarítmico, en contraste con el de aplicación paso a paso, de orden lineal. Las publicaciones [Martinez et al., 2006, 2007] presentan un diseño *hardware* del algoritmo de aplicación con *benchmark* de maximalidad empleando puertas lógicas, multiplexores, divisores, multiplicadores

y generadores de números aleatorios.

Algoritmo 2.4 Algoritmo de aplicación con *benchmark* de minimalidad [Fernández, 2006]. Donde $\phi_{M(R(U))}$ es el conjunto vacío, $w_{R(U)}$ es el conjunto de pares (regla, número de aplicación) que define la transición a la siguiente configuración, w representa al multiconjunto de objetos en la configuración actual, A es el conjunto de reglas de evolución e $input(r_i)$ indica los objetos consumidos por la regla r_i.

---

1. $w_{R(U,T)} \leftarrow \phi_{M(R(U))}$
2. **repetir**
3. $\quad r_i \leftarrow random(A)$
4. $\quad max \leftarrow maximalApplicable(r_i, w)$
5. $\quad$ **si** $max = 0$ **entonces**
6. $\quad\quad A \leftarrow A - \{r_i\}$
7. $\quad$ **sino**
8. $\quad\quad w' \leftarrow \phi_{M(U)}$
9. $\quad\quad$ **para todo** $r_j \in A - \{r_i\}$ **hacer**
10. $\quad\quad\quad w' \leftarrow w' + maximalApplicable(r_j, w) * input(r_j)$
11. $\quad\quad\quad min \leftarrow maximalApplicable(r_i, w - (w \bigcap w'))$
12. $\quad\quad\quad k \leftarrow random(min + 1, max)$
13. $\quad\quad\quad w_{R(U)} = w_{R(U)} + \{(r_i, k)\}$
14. $\quad\quad\quad w \leftarrow w - k * input(r_i)$
15. $\quad\quad$ **fin para**
16. $\quad$ **fin si**
17. **hasta** $|A| = 0$

Por último, en [Alonso et al., 2008] se describe el diseño *hardware* del sistema completo, con la integración de todos los módulos anteriormente desarrollados. También se ha presentado una implementación basada en microcontroladores [Gutiérrez et al., 2006, 2008], efectuando las modificaciones necesarias para adaptar el diseño a este tipo de dispositivos. El objetivo es crear un sistema distribuido de bajo coste, área y consumo. Los únicos resultados ofrecidos por el autor son pruebas de ejecución que no permiten valorar la potencia computacional lograda

con este enfoque, aunque propone como línea de futuro la experimentación con distintas arquitecturas.

El trabajo llevado a cabo por este grupo de investigación posee gran relevancia teórica, especialmente en los algoritmos de cálculo de número de aplicabilidad, que han sido adaptados por otros autores en implementaciones *software* [Martínez-del Amor et al., 2010]. En lo que respecta a los resultados de implementación, los autores han descrito diseños *hardware*, aunque sin ofrecer datos de rendimiento ni de implementación. No obstante, se observan algunas de las características no deseables presentes en el trabajo de Petreska.

En lo que respecta a la potencia de cómputo, el doble paralelismo inherente de los sistemas P no se ha implementado, únicamente existe un paralelismo a nivel de membranas, pero no a nivel de reglas para cada membrana. Además, el empleo de recursos *hardware* específicos y la ausencia de una metodología de trabajo origina una implementación con una limitada escalabilidad y extensibilidad y prácticamente ninguna flexibilidad. La inclusión de nuevas variedades o cambios provocaría un fuerte impacto en el *hardware* y a las instancias de la clase implementada, requiriendo la modificación de estructuras de datos, líneas de comunicación y elementos de procesado. En consecuencia, el trabajo realizado está más orientado al diseño de estructuras de datos y algoritmos, presentando diseños *hardware* genéricos no enfocados a la practicidad.

2.2.3.3. La plataforma de V. Nguyen: *Reconfig-P*

La línea seguida por V. Nguyen es una de las más actuales y completas en el área. Esta investigadora ha publicado varios trabajos que han culminado con la presentación de una tesis doctoral [Nguyen, 2010] en la que presenta dos posibles arquitecturas, *Reconfig-P* α y *Reconfig-P* β, basadas cada una de ellas en un enfoque distinto.

Reconfig-P α es un sistema orientado a reglas, esto es, las unidades básicas de cómputo representan las reglas de evolución de un sistema P (Fig. 2.6). Las regiones no son explícitamente implementadas, sino que son caracterizadas por sus multiconjuntos y reglas asociadas. Para cada membrana existe un vector en el que cada componente representa la multiplicidad de un tipo de objeto para esa

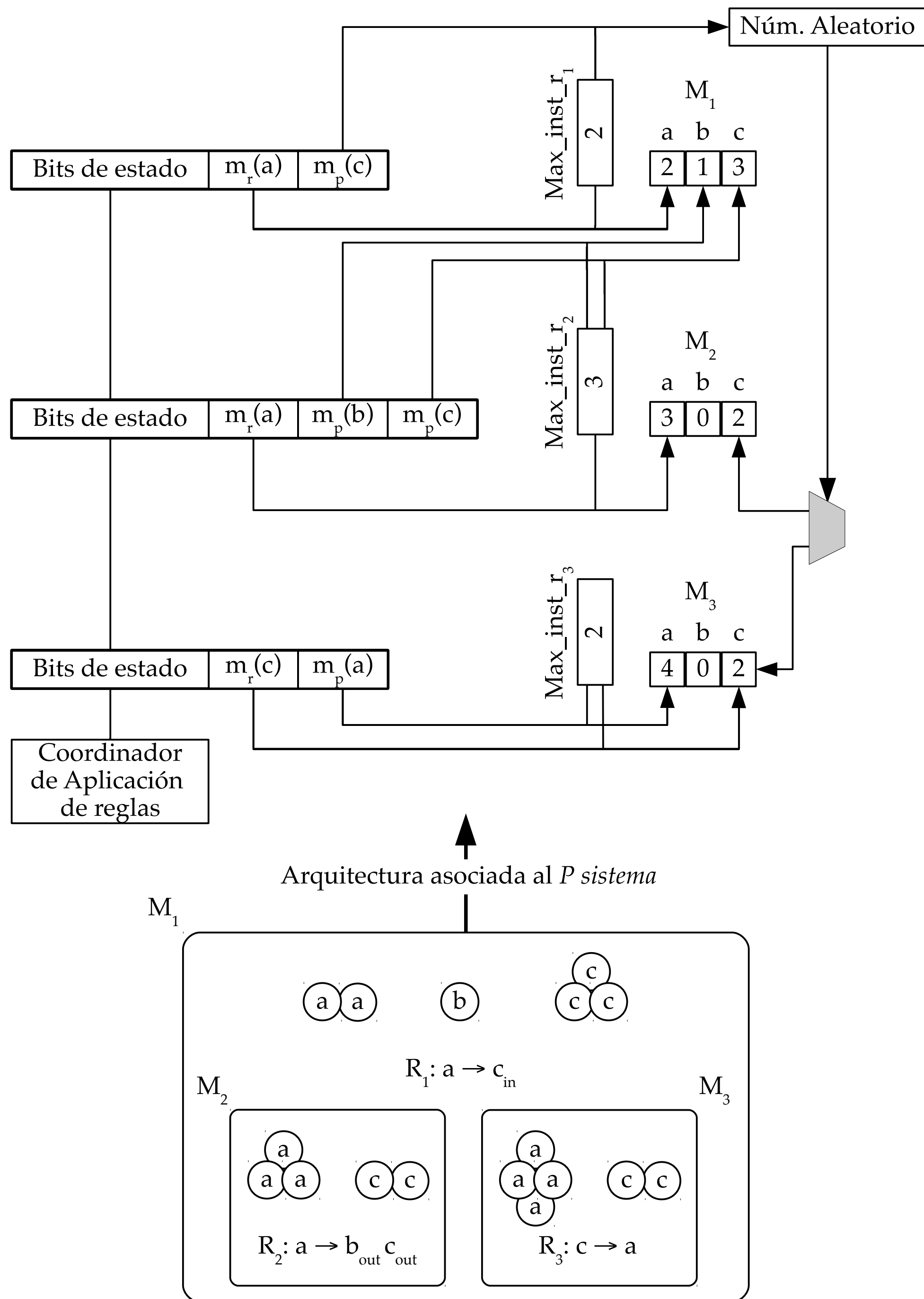


Fig. 2.6: Esquema de la arquitectura *Reconfig-P* α [Nguyen, 2010]. En ella, las reglas son aplicadas de forma determinista, empleando los bits de estado y el coordinador de reglas para la sincronización y control de la ejecución. Del mismo modo, se muestra un esquema de la representación de las reglas (elementos m_r y m_p) y de la estructura de datos asociadas a cada una de las membranas, M_i. Por último, la decisión de dónde generar el objeto c para la regla R_1, es implementada combinando un generador de números aleatorios y un multiplexor.

región. Las reglas presentan una estructura semejante, poseen tres bits de estado para tareas de sincronización y una componente por objeto participante, indicando en qué número es afectada su multiplicidad. La ejecución de las reglas se divide en dos fases: fase de preparación, donde se obtiene el número de aplicabilidad para cada regla; y fase de actualización, donde se produce la actualización de las multiplicidades de los objetos afectados. La sincronización del sistema es llevada a cabo por un módulo denominado Coordinador de aplicación de reglas, aplicando operaciones lógicas a los bits de estado de las reglas. Atendiendo a los conflictos de acceso a los vectores de multiplicidad de objetos, únicamente suceden en la fase de actualización, debido a que en la fase de preparación las prioridades determinan el orden de acceso. La autora ha planteado dos opciones para el acceso de escritura a los registros: modo orientado al espacio, en el que el registro es repetido el número de veces necesario como para que todas las reglas puedan escribir simultáneamente; y el modo orientado al tiempo, en el que los accesos son ordenados en el tiempo.

A diferencia del anterior, *Reconfig-P* β es orientado a regiones, por lo que las unidades básicas de procesado representan las regiones de un sistema P. Las membranas son implementadas como unidades de procesamiento paralelo, incluyendo las unidades de control y procesado de reglas, por lo que no existe una implementación directa de estas. Con respecto a los objetos, son implementados del mismo modo que en la versión α, empleando una matriz de registros por membrana. Considerando estos módulos básicos, la implementación del sistema está constituida por un conjunto de membranas (Fig. 2.7), consideradas las unidades de procesado del sistema, que a su vez se componen de dos módulos destinados a las fases de asignación de objetos (fase de preparación en *Reconfig-P* α) y a la de producción de objetos (fase de actualización en *Reconfig-P* α); y las matrices de registros que representan a los objetos. Todas las membranas están conectadas a una unidad de control encargada de la coordinación de las distintas regiones, y aquellas que poseen reglas de intercambio de objetos están comunicadas por canales. A excepción de los canales de comunicación ínter-membrana, en los que existe cierto acoplamiento por razones de optimización, las regiones son completamente independientes unas de otras, constituyendo unidades básicas.

Para mejorar la flexibilidad y extensibilidad de las arquitecturas, la autora ha implementado una herramienta *software* de generación automática denominada

P-Builder. A partir de un sistema P de entrada se genera el código de la implementación en lenguaje *Handel-C*, permitiendo seleccionar la versión de *Reconfig-P* y el modo de resolución de conflictos de escritura en registros. Su diseño está marcado por el patrón de diseño *Content-Form-Estrategy* diseñado por ella misma, y sus características más destacables son su modularidad y extensibilidad.

Reconfig-P es el primer sistema implementado en *hardware* que ofrece el doble paralelismo, a nivel de membranas y de reglas. Para la versión α los resultados obtenidos muestran una potencia superior a implementaciones *software* (de 14 a 500 veces más rápida) y *hardware* (de 2,5 a 31 veces más rápida) existentes y con una escalabilidad cercana a lineal en lo que respecta al tamaño del sistema. Atendiendo a la versión β, los resultados en ocupación *hardware* y escalabilidad son semejantes, con un incremento del 2 % en el peor caso, y una potencia inferior que comprende desde un 3 % hasta un 22 % dependiendo de las características del sistema. El tamaño máximo del sistema P que puede ser implementado en una FPGA Virtex II RC 2000 también ha sido incluido en los resultados, para *Reconfig-P* α el tamaño del sistema es de 450 reglas, 45 regiones y 135 tipos de objetos, mientras que para *Reconfig-P* β el tamaño del sistema es de 550 reglas, 55 regiones y 165 tipos de objetos.

La versión orientada a reglas presenta una potencia superior a la orientada a regiones, aunque esta optimización penaliza su extensibilidad y mantenimiento, especialmente en la implementación de estos tipos de sistemas.

En cuanto a las limitaciones de sus desarrollos, V. Nguyen únicamente ha implementado una versión determinista basada en prioridades de los sistemas P transitivos. La inclusión del no determinismo, de sistemas estocásticos, membranas activas y sistemas P con antiportadores/simportadores se plantean como posibles ampliaciones. Estas funcionalidades pueden comprometer las prestaciones de rendimiento y extensibilidad del sistema. En este sentido, la autora ha planteado un algoritmo para implementar el no determinismo denominado *Direct Nondeterministic Distribution* (DND), aunque aún no ha sido incorporado a la plataforma, por lo que no es posible valorar su impacto. Debido al área limitada de las FPGA, la escalabilidad podría aumentarse balanceando la carga computacional del sistema entre varios dispositivos, siempre y cuando el tiempo de comunicación sea mucho menor que el de procesado. Otra posibilidad consiste en hacer uso de la reconfigu-

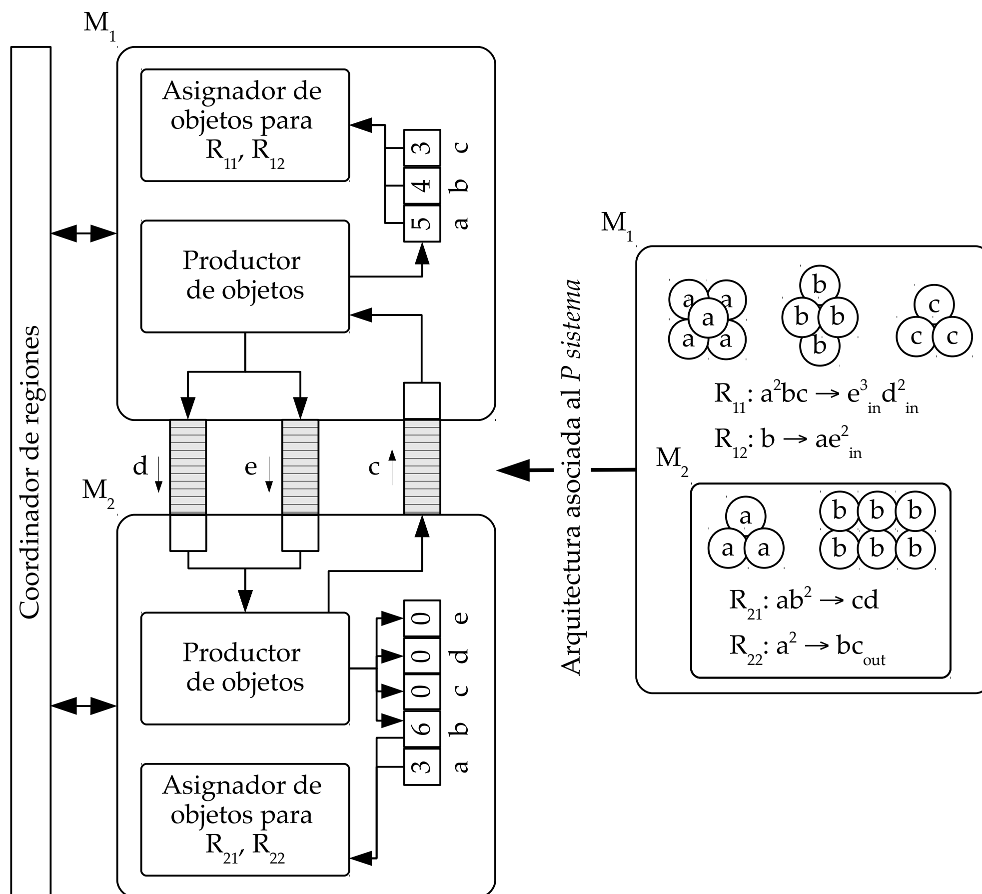


Fig. 2.7: Esquema de la arquitectura *Reconfig-P* β [Nguyen, 2010]. En ella, se toman las regiones como elementos centrales. Así, es preciso un coordinador de regiones que se ocupe de la sincronización entre los distintos módulos, que representan las regiones, y en los se implementa la lógica asociada a las reglas, no implementadas de forma directa.

rabilidad dinámica de estos dispositivos, aunque su utilidad sería dependiente de las características del sistema.

2.3. Conclusiones

En este capítulo se han introducido los sistemas de membranas, presentándolos como un modelo de computación con utilidad práctica en la actualidad. Del mismo modo se han descrito las implementaciones existentes y las tecnologías empleadas, así como los requisitos deseables en estas: potencia de cómputo, flexibilidad y escalabilidad.

Las implementaciones *software* secuenciales y multihilo, caracterizadas por una alta flexibilidad y extensibilidad, fueron las primeras en ser desarrolladas. No obstante, debido a su limitada potencia de cómputo y escalabilidad son inadecuadas para sistemas relativamente grandes. Destacan las herramientas P-Lingua y MetaPlab, que ofrecen un entorno completo de desarrollo y ejecución de sistemas P, permitiendo la inclusión de nuevas implementaciones y funcionalidades como módulos. Con respecto al uso de arquitecturas distribuidas, no han proporcionado resultados satisfactorios. Prueba de ello son los elevados tiempos de comunicación en comparación con los de cómputo. Por último, ante las limitaciones de las implementaciones *software*, gran parte del esfuerzo se está derivando al desarrollo de implementaciones basadas en el uso de GPU, desarrollando implementaciones *ad-hoc* para modelos específicos, donde consiguen superar al resto de implementaciones *software*, constituyendo un área en proceso de consolidación.

Las implementaciones *hardware* persiguen obtener mayor potencia de cómputo y escalabilidad, manteniendo una flexibilidad y extensibilidad adecuadas. En este área existen tres líneas de investigación: B. Petreska, pionera en esta rama, demuestra que es posible el uso de FPGA para este fin; el Grupo de Computación Natural de la UPM ha presentado avances en el campo de la algoritmia y las estructuras de datos; y el trabajo de V. Nguyen constituye el primer intento por desarrollar una implementación de alta potencia y escalabilidad en *hardware* reconfigurable.

La implementación de V. Nguyen presenta una potencia y escalabilidad adecuadas. Sin embargo, únicamente soporta un sistema P muy básico y, aunque la autora ha planteado un algoritmo para soportar el no determinismo, no ha demostrado la

posibilidad de incorporarlo sin comprometer significativamente la potencia, flexibilidad y/o escalabilidad de su solución. Esta limitada extensibilidad es una fuerte restricción a la hora de implementar un modelo de computación orientado a máquinas y del que hay gran cantidad de variedades. Adicionalmente, desde un punto de vista teórico, estas implementaciones se basan en el modelo presentado en el artículo fundacional [Păun, 2000], por lo que sufren restricciones muy fuertes relacionadas con la topología, el tipo de objetos y las reglas del sistema P. Por tanto, es importante contar con implementaciones más flexibles y, en este sentido, los resultados teóricos de [Freund and Verlan, 2007] pueden suponer un buen punto de partida de cara a una implementación que cubra la mayor parte de las restricciones acerca de estos aspectos (topología, reglas y objetos). En consecuencia, son necesarias nuevas arquitecturas que presenten una extensibilidad mejorada o que al menos implementen un mayor conjunto de sistemas P, así como su posible integración con entornos de desarrollo, como P-Lingua.

En conclusión, el diseño *hardware* de este tipo de implementaciones presenta dos retos importantes. Por un lado el paralelismo inherente de este tipo de modelos, considerando el no determinismo y los modos de derivación formales. Por otro lado, el ser un modelo de computación orientado a máquinas obliga a generar una máquina por cada instancia del problema, lo que hace prácticamente imposible utilizar *hardware* no reconfigurable, como por ejemplo *hardware* específico o ASIC. En este sentido, las FPGA suponen la única alternativa *hardware* viable de cara a la implementación de este tipo de modelo computacional, tal y como han demostrado los trabajos realizados en el área.

Parte II

Desarrollo de la arquitectura Almond PS y del *software* de generación

Capítulo 3

Arquitectura Almond PS

Una vez que se han presentado las nociones básicas acerca de los sistemas P, sus aplicaciones y los trabajos previos en el área de su implementación, en este capítulo se presenta la solución aportada por el autor: la arquitectura Almond PS. Se trata de una arquitectura de ejecución de sistemas P no deterministas basados en tejidos, los cuales realizan la selección de reglas siguiendo una distribución uniforme. El punto de partida consiste en, para cada transición del sistema, representar el conjunto de multiconjuntos de reglas aplicables, $Appl(\Pi, C, \delta)$, como un lenguaje no ambiguo y libre de contexto, para seleccionar a continuación una palabra (multiconjunto) del lenguaje ($Appl(\Pi, C, \delta)$). En los fundamentos teóricos de la implementación, se presenta un método con el que calcular, empleando series formales de potencias, el número de palabras del lenguaje ($|Appl(\Pi, C, \delta)|$), sin necesidad de generar todas las palabras que lo forman. A continuación, se describe otro procedimiento para generar de forma directa la palabra n-ésima del lenguaje. Por lo tanto, únicamente queda generar, de forma no determinista, un número aleatorio que siga una distribución equiprobable comprendido entre 0 y el tamaño del lenguaje, y a continuación generar la palabra correspondiente. Es preciso destacar que el punto de vista empleado en el diseño es ligeramente distinto al de otros trabajos anteriores, permitiendo alcanzar un rendimiento cercano al ideal. Aunque esto implica una cierta pérdida de flexibilidad, al reducir el rango de entrada de sistemas P, es importante aclarar que la validez para un sistema P concreto no depende de su clase, como sí ocurre en otros trabajos, sino de la complejidad de

las dependencias entre sus reglas; lo que permite aceptar como entrada de la arquitectura desarrollada un amplio rango de sistemas P. Como resultado, se obtiene una implementación *hardware* basada en FPGA, con un rendimiento aproximado de 2×10^7 transiciones por segundo, independientemente de la cantidad de reglas empleadas o del tipo de objetos.

En este capítulo se describen brevemente los problemas más destacados a la hora de afrontar la implementación de un sistema P, así como las distintas soluciones aportadas por los autores de los trabajos anteriores, y los fundamentos matemáticos introductorios necesarios para la comprensión y desarrollo de la solución propuesta por el doctorando. A continuación, se presentan los fundamentos teóricos sobre los que se sustenta la implementación propuesta, seguidos por la formalización de los sistemas P de entrada aceptados. Seguidamente, se detalla la arquitectura *hardware* correspondiente, a la que se ha denominado *Almond PS* y, por último, se presentan las conclusiones.

3.1. Evolución

El diseño de la arquitectura Almond PS persigue la filosofía de implementar una arquitectura de ejecución de sistemas P priorizando las características de la tecnología FPGA frente a las de los modelos teóricos. Esto conlleva la búsqueda de nuevos algoritmos, con el objetivo de optimizar la potencia computacional de la implementación, al tiempo que se mantienen unos parámetros adecuados de escalabilidad y extensibilidad.

3.1.1. La raíz del problema

Tomando como punto de partida el entorno formal de redes de células presentado en el anterior capítulo (Sección 2.1.1), la ejecución de un sistema P se reduce a, partiendo de una configuración inicial C_0, calcular y aplicar sucesivas transiciones hasta alcanzar una condición de parada (Fig. 3.1). A su vez, cada transición se puede dividir en una Fase de Selección en la que, tomando como configuración origen C_i, se selecciona un conjunto R del conjunto $Appl(\pi, C_i, \delta)$; y una Fase de Aplicación, en la que se aplican las reglas del conjunto R a la configuración C_i,

obteniendo de ese modo la próxima configuración, o C_{i+1}. La Fase de Aplicación consiste en la ejecución de operaciones aritméticas básicas: multiplicación, suma y resta. Adicionalmente, se debe contemplar una persistencia de los resultados obtenidos en cada transición.

La Fase de Selección es el proceso crítico de la ejecución, la que diferencia a las distintas soluciones, del mismo modo en que es una de las principales diferencias entre las variantes de sistemas P. En este sentido confluyen dos aspectos: el modo de derivación, δ, y la selección, determinista o no, del multiconjunto R, que determinan en gran parte la semántica del modelo. El principal desafío consiste en que, a efectos prácticos, el conjunto $Appl(\pi, C_i, \delta)$ es demasiado extenso como para ser calculado en cada transición. Por ello, las implementaciones deben resolver este problema haciendo uso de algoritmos que, manteniendo la semántica del modelo, eviten la generación del conjunto.

3.1.2. Resolución de otros autores

El problema de selección de reglas puede reducirse a un problema de distribución de recursos entre unidades consumidoras, en el que se puede aplicar un algoritmo de búsqueda con un espacio asociado, contemplándose soluciones finales y efectivas o candidatas, según verifiquen o no el modo de derivación δ del modelo, capturando de ese modo la semántica del sistema P.

En este sentido, en la bibliografía se contemplan dos enfoques: directo e iterativo. En el primero el multiconjunto R es obtenido en un único paso atómico. El algoritmo teórico descrito anteriormente, no aplicable en la práctica, corresponde a este grupo. Otra alternativa consiste en generar de forma aleatoria una solución candidata, y posteriormente comprobar si verifica las restricciones del modo de derivación δ, repitiendo el proceso hasta llegar a una solución final. Este segundo algoritmo, al no requerir el cálculo del espacio $Appl(\pi, C_i, \delta)$, es abordable desde un punto de vista práctico, aunque presenta problemas de eficiencia e incertidumbre acerca del número de pasos requeridos para llegar a una solución final, por lo que debe tomarse como un suelo en términos de eficiencia para el diseño de cualquier algoritmo.

La gran multitud de los algoritmos desarrollados abordan un enfoque iterativo.

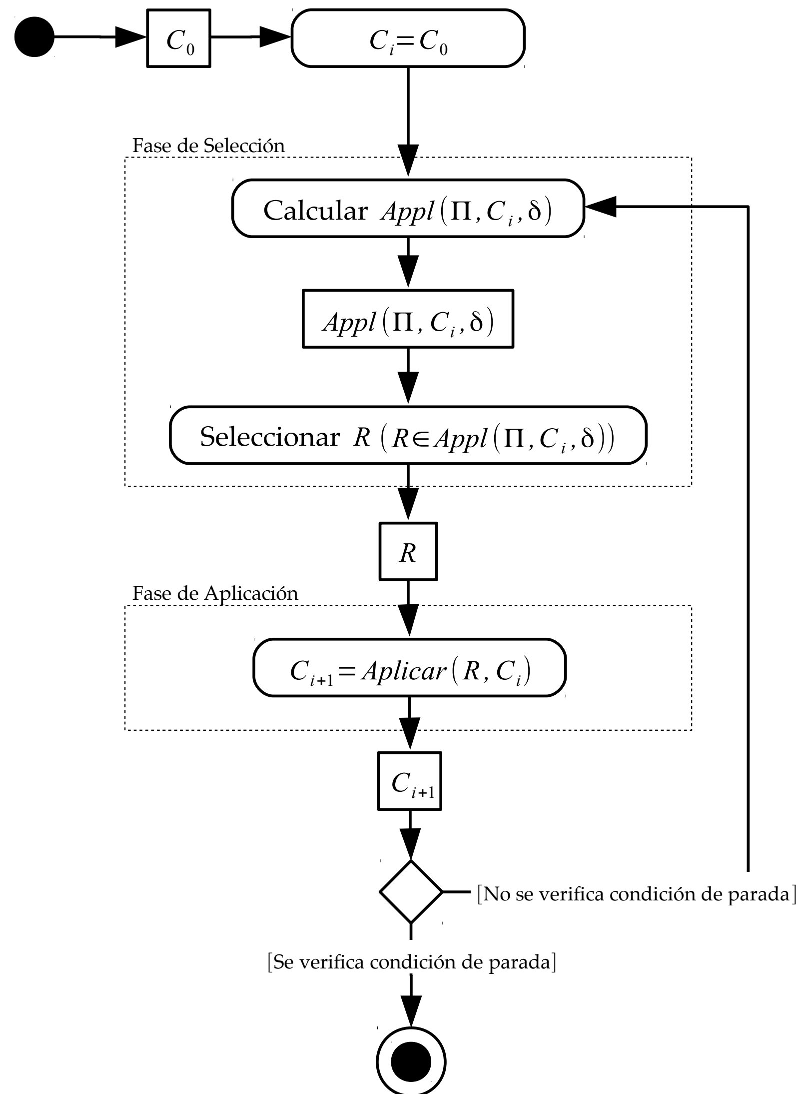


Fig. 3.1: Diagrama de actividad UML que muestra la división en funcionalidad de la ejecución de un sistema P.

De ese modo, una transición es calculada en varios pasos de forma, como ya se ha indicado, iterativa. En cada paso, se obtiene una solución candidata iterando, de un modo no determinista, sobre cada regla del sistema, aumentando su número de aplicación en una unidad [Fernández, 2006], o en un número mayor que uno, dependiendo del algoritmo concreto, aunque los límites inferior y superior del intervalo están comprendidos, respectivamente, entre la unidad y la aplicabilidad máxima de la regla en la configuración origen, C_i [Fernández, 2006; Nguyen et al., 2009; Cardona et al., 2010; Martínez-del Amor et al., 2010; Martínez-del Amor et al., 2011]. Estos algoritmos convergen siempre hacia una solución final, descartando en cada decisión un conjunto de soluciones candidatas. No obstante, a priori el número de iteraciones requeridas para converger a una solución final no es determinista, por lo que generalmente los autores acotan a un número de iteraciones la ejecución del algoritmo: desde un único paso en [Nguyen et al., 2009; Martínez-del Amor et al., 2010; Martínez-del Amor et al., 2011], a un número máximo que recibe el sistema como entrada [Cardona et al., 2010]. En estos casos, esta restricción puede generar una solución candidata que no verifique el modo de derivación δ, por lo que es necesario un post-procesado de orden lineal respecto al número de reglas, para obtener una solución final.

Como puede observarse, las soluciones desarrolladas toman como enfoque inicial los sistemas P, utilizando algoritmos que, si bien pueden mejorar considerablemente su tiempo de ejecución empleando arquitecturas paralelas, no han sido concebidos priorizando las características de estas últimas. Este hecho es agravado por la propia naturaleza de los sistemas P, en la que existe un número enorme de dependencias entre reglas, motivado por la competencia por los recursos comunes, además de los problemas de acceso en escritura al persistir los resultados obtenidos.

En este sentido, uno de los algoritmos más recientes, empleado en la implementación de un PDP [Colomer et al., 2010] y detallado en [Martínez-del Amor et al., 2013], rompe esta tendencia. Tanto el algoritmo, como las estructuras de datos empleadas guardan una estrecha relación con la tecnología empleada, CUDA. El algoritmo requiere una agrupación de reglas en conjuntos disjuntos basándose en un conjunto de restricciones. De ese modo, los problemas de dependencias entre reglas han sido parcialmente resueltos con una distribución equitativa y determinista de los recursos entre los grupos, para posteriormente repartirlos entre las

reglas de cada conjunto siguiendo una distribución multinomial, aplicando algunas correcciones para lograr verificar las restricciones del modo de derivación δ. En consecuencia, se ha eliminado gran parte del no determinismo inherente del modelo, para optimizar la ejecución del algoritmo en la tecnología objetivo.

Respecto a las implementaciones de sistemas P empleando *hardware* reconfigurable, han tomado como punto de partida la descripción teórica del modelo. Este hecho es claramente apreciable en la primera implementación de B. Petreska [Petreska and Teuscher, 2004], en la que existe una correspondencia directa entre las entidades del modelo teórico y las del propio sistema *hardware*. Esta relación compromete la potencia, escalabilidad y flexibilidad del sistema, a favor de una mejor comprensión de su funcionamiento respecto al modelo teórico.

V. Nguyen toma como punto de partida el trabajo de B. Petreska en [Petreska and Teuscher, 2004], estableciendo la necesidad de romper con este enfoque en aras de conseguir un sistema *hardware* de mejor calidad, según los tres aspectos mencionados (Sec. 2.2.1). Sin embargo, aunque su avance en este sentido es destacado, no logra romper por completo esta correspondencia. Así, su primera implementación, Reconfig-P α, es descrita como una implementación orientada a reglas [Nguyen et al., 2008], mientras que Reconfig-P β es calificada como orientada a regiones o membranas [Nguyen et al., 2010].

Aunque los algoritmos desarrollados por otros autores han conseguido mejorar los tiempos de ejecución, logrando implementaciones cada vez más eficientes, es necesario aportar nuevas soluciones que reduzcan los tiempos de ejecución de las implementaciones existentes de sistemas P. Como respuesta a esta necesidad, se ha propuesto un nuevo enfoque, que toma como pilar fundamental de diseño priorizar las características tecnológicas de las soluciones propuestas frente al modelo teórico de los sistemas P objetivo. En este sentido, ha sido necesario contar con una modelización lo más abstracta posible de los sistemas P, tomándose como punto de partida el entorno formal descrito por R. Freund y S. Verlan en [Freund and Verlan, 2007], y comentado en el capítulo anterior (Sección 2.1.1).

3.1.3. Gramáticas libres de contexto como generadores de series formales de potencias

El fundamento teórico empleado en el diseño de la arquitectura Almond PS está basado en la teoría de series formales de potencias, especialmente en relación a la teoría de lenguajes formales. Es por ello que se introduce en este apartado, sugiriendo la lectura de [Rozenberg and Salomaa, 1997] en caso de que el lector desee profundizar en la materia.

Considerando un lenguaje, $L(G)$, como un conjunto de secuencias de caracteres, podemos definir su gramática, G, como el conjunto de reglas que enumera el conjunto de cadenas que pertenecen al lenguaje. Cada cadena o palabra está formada por caracteres o símbolos, cuyo conjunto se denomina alfabeto del lenguaje. Para los siguientes párrafos, denotaremos mediante $|w|$ la longitud de la palabra w o la cardinalidad del multiconjunto (o conjunto) w.

Definición 9. *Se define una gramática libre de contexto como una* $4 - tupla$*:*

$$G = (V_N, V_T, S, P)$$

donde:

V_N *Es un conjunto de símbolos no terminales,*

V_T *Es un conjunto de símbolos terminales,*

S *Se corresponde con el símbolo inicial,*

P *Es el conjunto de reglas generadoras del lenguaje, que deben ser de la forma*

$$V_N \rightarrow (V_N \cup V_T)^*$$

Dada una gramática libre de contexto $G = (V_N, V_T, S, P)$, y una regla generadora $(x_i \rightarrow \alpha_j) \in P$, se define como parte izquierda de la regla a la cadena x_i, mientras que α_j se corresponde con su parte derecha. En este caso, $x_i \in V_N$ y $\alpha_j \in (V_N \cup V_T)$. Es preciso notar que varias reglas pueden tener en común su parte

izquierda, por lo que la generación no siempre es determinista, siendo no determinista en aquellas ocasiones en las que existan dos o más reglas con la misma parte izquierda, presentando un grado de ambigüedad estructural que se corresponde con el número de resultados posibles.

Es fácil observar que, siendo $L(G)$ el lenguaje generado por la gramática G, su alfabeto, A, se corresponde con los símbolos terminales V_T de G. Si se considera A^* como el conjunto de todas las cadenas generadas a partir de V_T (un monoide libre generado por G), es posible definir un lenguaje (asociado al alfabeto A) como un subconjunto de A^*.

A continuación, se considera una serie formal de potencias f como una aplicación $f : A^* \to \mathbb{N}$, donde $\mathbb{N}$ es el conjunto de los enteros no negativos (en el caso general, una serie formal de potencias es una aplicación de un monoide libre en un semianillo). Dicha aplicación se escribe habitualmente como:

$$f = \sum_{w \in A^*} f(w)w. \tag{3.1}$$

Una gramática libre del contexto $G = (V_N, V_T, S, P)$ puede considerarse como un conjunto de ecuaciones $x_i = \alpha_1 + \cdots + \alpha_{n_i}$, para cada x_i no terminal de G, donde α_j son las partes derechas de las producciones $x_i \to \alpha_j$, $1 \leq j \leq n_i$, siendo n_i el número de reglas con igual parte izquierda x_i. Una solución de G es un conjunto de series formales de potencias $s_1, \ldots, s_k$, tal que la sustitución de x_i por s_i en las ecuaciones anteriores las convierte en la identidad, es decir, las correspondientes series son iguales término a término. Se sabe que $s_i = \sum_{w \in A^*} f_i(w)w$ ([Chomsky and Schützenberger, 1963]), donde $f_i(w)$ es el número de derivaciones distintas de la parte izquierda de w empezando por x_i. Considerando la aplicación que relaciona los elementos del conjunto A hacia el mismo símbolo, por ejemplo x, obtenemos las funciones generadoras para un x_i no terminal:

$$f_i = \sum_{n=0}^{\infty} \sum_{|w|=n} f_i(w)x^n.$$

Considerando $f_i(n) = \sum_{|w|=n} f_i(w)$, la ecuación anterior puede ser reescrita

como:

$$f_i = \sum_{n=0}^{\infty} f_i(n)x^n.$$

Supongamos que $x_1 = S$, donde S es el símbolo inicial de G. Entonces f_1 se denomina la función generadora de G. Si G no es ambigua, entonces $f_1(n)$ proporciona el número de palabras de longitud n de G (denotamos como $[x^n]f$ al coeficiente n-ésimo de f, es decir, $[x^n]f = f(n)$).

Sea ϕ el morfismo definido por

$$\begin{aligned} \phi(\lambda) &= 1, \\ \phi(a) &= x \qquad \forall a \in V_T, \\ \phi(x_i) &= f_i \qquad x_i \in V_N. \end{aligned}$$

Sea $x_i \to v_{i1} \mid \cdots \mid v_{ik}$ el conjunto de producciones asociadas a x_i. Entonces f_i puede calcularse como la solución del siguiente sistema de ecuaciones:

$$f_i = \sum_{j=1}^{k} \phi(v_{ij}). \tag{3.2}$$

Ejemplo 1. *Considerando una gramática de ejemplo $G = (V_N, V_T, S, P)$, donde:*

$$\begin{aligned} V_N &= (S), \\ V_T &= (a, b), \\ S &= (S), \\ P &= (S \to SbS; S \to a). \end{aligned}$$

El conjunto de ecuaciones asociado a cada símbolo no terminal x_i de G se corresponde con

$$S = a + SbS$$

Considerando la sustitución $\psi(s) = a + sbs$, donde s es una serie de potencias,

se obtiene la siguiente secuencia infinita $\varrho_0, \varrho_1, \ldots$*:*

$$\begin{array}{lllllll}
\varrho_0 & = & s_0 & = & 0 & & \\
\varrho_1 & = & s_1 & = & a + s_0 b s_0 & = & a \\
\varrho_2 & = & s_2 & = & a + s_1 b s_1 & = & a + aba \\
\varrho_3 & = & s_3 & = & a + s_2 b s_2 & = & a + (a + aba)b(a + aba) \\
 & & & & & = & a + aba + ababa + ababa + abababa \\
 & & & & & = & a + aba + 2ababa + abababa \\
\varrho_4 & = & s_4 & = & a + s_3 b s_3 & = & a + aba + (ab)^2a + 5(ab)^3a + 6(ab)^4a + \\
 & & & & & & 6(ab)^5a + 4(ab)^6a + (ab)^7a \\
\ldots & \ldots & & & & &
\end{array}$$

De ese modo, es posible definir s_∞ *como la siguiente serie de potencias:*

$$s_\infty = \sum_n \binom{2n}{n} \frac{1}{n+1} (ab)^n a = a + 1(ab)^1 a + 2(ab)^2 a + 5(ab)^3 a + \ldots$$

Sustituyendo en (3.1)*, se obtiene:*

$$f_i(w) = \binom{2n}{n} \frac{1}{n+1}$$

Correspondiendo $f_i(w)$ *al grado de ambigüedad estructural de la cadena* w*. Del mismo modo, el coeficiente i-ésimo de* s_∞ *representa el grado de ambigüedad estructural de las cadenas de longitud* i*. Así, tomando la cadena* $w = ababa$*, puede haberse originado de dos formas:* $(ab(aba))$ *o* $((aba)ba)$*.*

Para una gramática regular G, el sistema (3.2) se hace lineal [Chomsky and Schützenberger, 1963]. Considerando un autómata finito $\mathcal{A} = (V, \mathbb{Q}, q_0, Q_f, \delta)$ equivalente a G, se obtiene que el sistema (3.2) corresponde al siguiente sistema (recuérdese que x se considera constante)

$$Q = xMQ + F. \tag{3.3}$$

donde,

- $Q = [q_1 \ldots q_n]^t$, $q_i \in \mathbb{Q}$, $1 \le i \le n$ es el vector que contiene todos los estados.

- $F = [a_0 \dots a_n]^t$, es el vector característico de estado final, *i.e.* , $a_i = 1$ si q_i es un estado final y cero en otro caso.

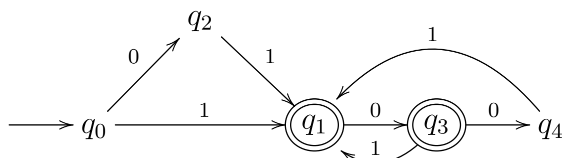
- M es la matriz de transferencia del autómata $\mathcal{A}$, *i.e.* , la matriz de incidencias del grafo representado por $\mathcal{A}$ con los valores negativos reemplazados por cero.

Se debe remarcar que en el caso de un lenguaje regular también es posible contabilizar el número de palabras de longitud n sumando las columnas correspondientes a los estados finales de la potencia n-ésima de la matriz de transferencia del autómata correspondiente:

$$f_i(n) = \sum_{q_j \in Q_f} (M^n)_{i,j}.$$

Es conocido que la función generadora f para un lenguaje regular es racional. Esto implica la existencia de una recurrencia finita $f(n) = \sum_{j=1}^{k} a_j f(n-j)$, $k > 0$, $a_j \in \mathbb{Z}$ que se mantiene para n grande.

Ejemplo 2. *Considerando el lenguaje regular L_I reconocido por el siguiente autómata*



Entonces el vector característico de estado final F para este autómata viene definido por $F = [0, 1, 0, 1, 0]^t$ y la función de transferencia M por

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

El correspondiente sistema (3.3) *de ecuaciones lineales tiene la siguiente solu-*

ción

$$q_0 = \frac{x^3 + 2x^2 + x}{1 - x^2 - x^3},$$
$$q_1 = \frac{x + 1}{1 - x^2 - x^3},$$
$$q_2 = \frac{x^2 + x}{1 - x^2 - x^3},$$
$$q_3 = \frac{x^2 + x + 1}{1 - x^2 - x^3},$$
$$q_4 = \frac{x^2 + x}{1 - x^2 - x^3}.$$

Es posible expandir q_0 *para obtener* $q_0(n)$ *(*$= [x^n]q_0$*),*

$$q_0 = x + 2x^2 + 2x^3 + 3x^4 + 4x^5 + 5x^6 + 7x^7 + 9x^8 + \dots$$

Los coeficientes de las series anteriores proporcionan el número de palabras para la longitud correspondiente. Por ejemplo, existen 9 *palabras de longitud* 8 *en* L_I*.*

No es difícil verificar que los coeficientes obtenidos $[x^n]q_k$*,* $0 \leq k \leq 4$*, para las series de potencias correspondientes son casos particulares de la secuencia de Padovan [Weisstein, 2014]* $q_k(n) = q_k(n-2) + q_k(n-3)$*,* $n > 3$*, con los siguientes valores iniciales:*

k	$q_k(0)$	$q_k(1)$	$q_k(2)$
0	1	1	2
1	1	1	1
2	0	1	1
3	1	1	2
4	0	1	1

3.2. Fundamentos teóricos. Parte formal

Como se ha mencionado anteriormente, existen varias características de un sistema P estático que lo definen: topología, tipo de objetos, reglas, modo de

derivación y elección del multiconjunto de reglas de aplicación R del conjunto *Appl*.

El entorno descrito en [Freund and Verlan, 2007] presenta una abstracción de los sistemas P, reduciendo la complejidad de su topología, tipos de objetos y reglas. Los dos primeros se modelan empleando una codificación que relaciona tipos de objetos, membranas y otros elementos que completan la semántica del modelo del sistema P original, *i.e.* las cargas en los sistemas P estocásticos empleados en [Cardona et al., 2010]. Atendiendo a las reglas, estas son abstraídas por otras con una semántica normalizada. Así, el sistema P original puede ser representado empleando un grafo de dependencia junto con el modo de derivación y el modo de selección del conjunto R para cada transición.

Para esta sección, consideraremos un sistema P estático, Π, de cualquier tipo, que evoluciona según cualquier modo de derivación, una vez elegido un multiconjunto de reglas de manera no determinista.

La idea principal de cara a la construcción de una arquitectura de ejecución que sea rápida, es evitar el cálculo del conjunto $Appl(\Pi, C, \delta)$, y computar directamente el multiconjunto de reglas a aplicar. El objetivo es el de aprovechar las ventajas de la tecnología FPGA para efectuar ese cálculo en el menor tiempo posible. Para ello se hará uso de las funciones $NBVariants(\Pi, C, \delta)$ y $Variant(n, \Pi, C, \delta)$. Estas funciones han sido diseñadas e implementadas en *hardware* tomando como base los conceptos comentados en la sección anterior.

Definición 10. *Sea* $\Pi = (n, V, w, Inf, R)$ *un sistema P con un modo de derivación* δ*,* C *una configuración válida de* Π*, y* $Appl(\Pi, C, \delta)$ *el conjunto de todos los multiconjuntos de reglas* $r \in R$ *aplicables en la configuración* C*. Se define* $NBVariants(\Pi, C, \delta)$ *como la cardinalidad del conjunto* $Appl(\Pi, C, \delta)$*.*

Definición 11. *Sea* $\Pi = (n, V, w, Inf, R)$ *un sistema P con un modo de derivación* δ*,* C *una configuración válida de* Π*, y* $Appl(\Pi, C, \delta)$ *el conjunto de todos los multiconjuntos de reglas* $r \in R$ *aplicables en la configuración* C *ordenado siguiendo una enumeración inicial fija. Se define* $Variant(n, \Pi, C, \delta)$*, donde* $1 \leq n \leq NBVariants(\Pi, C, \delta)$*, como el multiconjunto de reglas correspondiente al elemento* n*-ésimo de* $Appl(\Pi, C, \delta)$*.*

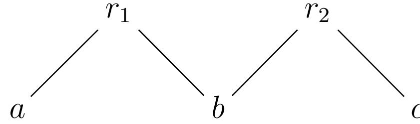
Es obvio que si cada función es calculada en tiempo constante, entonces el multiconjunto de reglas a aplicar también puede calcularse en tiempo constante.

A continuación se introduce el concepto de grafo de dependencia:

Definición 12. *Sea* $\Pi = (n, V, w, Inf, R)$ *un sistema P estático, se define un grafo de dependencia como un grafo bipartito ponderado,* $G = (U, W, E)$*, donde la primera partición contiene un nodo etiquetado con* x_i *por cada objeto* x_i *de* Π*,* i.e. $U = V$*, mientras que la segunda partición contiene un nodo etiquetado con* r_j *por cada regla* r_j *de* Π*,* i.e. $W = R$*. Así, existirá una arista entre un nodo* $r_j \in W$ *y un nodo* $x_i \in U$ *ponderada con un peso* k*,* $(r_j, x_i; k) \in E$*, si* $x_i^k \in lhs(r_j)$ *(y* $x_i^{k+1} \notin lhs(r_j)$*).*

En los siguientes párrafos se discutirán diferentes métodos para la construcción de las funciones $NBVariants(\Pi, C, \delta)$ y $Variant(n, \Pi, C, \delta)$, para diferentes clases de sistemas P y tomando como base sus grafos de dependencia.

Ejemplo 3. *Considerando un sistema P,* Π_1*, con las reglas* $r_1 : ab \to u$ *y* $r_2 : bc \to v$*. Estas reglas definen el siguiente grafo de dependencia:*



Sean N_a, N_b *y* N_c *el número de objetos a, b y c en una configuración* C*. Se define* N_1 *como el número máximo de aplicación de la regla* r_1*,* N_2 *como el número máximo de aplicación de la regla* r_2*, y* N *como el mayor número de aplicaciones común entre* r_1 *y* r_2*. Formalmente,*

$$N_1 = \text{mín}(N_a, N_b),$$
$$N_2 = \text{mín}(N_b, N_c),$$
$$N = \text{mín}(N_1, N_2).$$

Suponiendo que Π *evoluciona en un modo de derivación paralelo maximal. El conjunto* $Appl(\Pi, C, max)$ *puede calcularse como sigue:*

$$Appl(\Pi_1, C, max) = \bigcup_{p+q=N} \left\{ r_1^{p+k_1} r_2^{q+k_2} \right\},$$

donde $k_j = N_j \ominus N$, $1 \leq j \leq 2$, *donde* $\ominus$ *es la operación de resta positiva*[1].

Con esta representación está claro que $NBVariants(\Pi_1, C, max) = N + 1$, *el cual puede computarse en tiempo constante sobre FPGA.*

La función $Variant(n, \Pi_1, C, max)$ *puede definirse como el elemento* n*-ésimo dentro del orden lexicográfico de los elementos de* $Appl(\Pi_1, C, max)$ *y tiene la siguiente fórmula*

$$Variant(n, \Pi_1, C, max) = r_1^{N-n-1+k_1} r_2^{n-1+k_2}.$$

Nótese que la fórmula anterior también puede ser computada en tiempo constante utilizando una FPGA.

El siguiente paso consiste en tener un método que permita obtener las funciones $NBVariants$ y $Variant$.

Respecto a la función $NBVariants$, esta podría ser obtenida empleando series formales de potencias. De ese modo, se observa que el lenguaje $\cup_{N>0} L_N$, donde $L_N = \{r_1^p r_2^q \mid p + q = N\}$, es regular. Además, esto implica que $L_N = r_1^* r_2^* \cap A^N$, siendo A el alfabeto $\{r_1, r_2\}$. A continuación se muestra el autómata A_1 para el lenguaje $r_1^* r_2^*$.

(3.4)

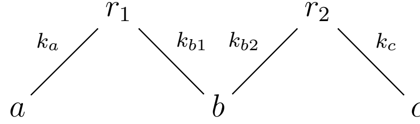
La matriz de transferencia de este autómata es $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ y el vector característico de estado final es $[1, 1]^t$. Utilizando la ecuación (3.3) se obtiene la función generadora para L_N: $q_0 = \frac{1}{(1-x)^2}$. Es fácil comprobar que $[x^n] q_0 = n + 1$.

[1] La resta positiva de dos números reales x, y se define como:

$$x \ominus y = \begin{cases} x - y, & \text{si } x \geq y, \\ 0, & \text{si } x < y \end{cases}$$

A continuación, se modificará el ejemplo anterior considerando reglas ponderadas.

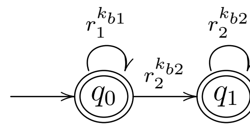
Ejemplo 4. *Considerando un sistema P,* Π_2*, con dos reglas* $r_1 : a^{k_a}b^{k_{b1}} \to u$ *y* $r_2 : b^{k_{b2}}c^{k_c} \to v$*. Estas reglas definen el siguiente grafo de dependencia:*



Sean N_a, N_b *y* N_c *el número de objetos a, b y c en una configuración* C*. Se define* N_1 *como el número máximo de aplicación de la regla* r_1*,* N_2 *como el número máximo de aplicación de la regla* r_2*,* N *como el mayor número de aplicaciones común entre* r_1 *y* r_2*, y* $\bar{N}$ *como el número de elementos b que será disputado entre las reglas* r_1 *y* r_2*. Formalmente,*

$$\begin{aligned} N_1 &= \text{mín}([N_a/k_a], [N_b/k_{b1}]), \\ N_2 &= \text{mín}([N_b/k_{b2}], [N_c/k_c]), \\ N &= \text{mín}(N_1, N_2), \\ \bar{N} &= \text{mín}(k_{b1}N_1, k_{b2}N_2). \end{aligned}$$

Suponiendo que Π_2 *evoluciona según un modo de derivación paralelo maximal,* A_2 *es el autómata de reconocimiento del lenguaje* $(r_1^{kb1})^*(r_2^{kb2})^*$*.*



Sea $L'_N = A_2 \cap A^N$ *(*$A = \{r_1, r_2\}$*), se tiene que*

$$Appl(\Pi_2, C, max) = \bigcup_{pk_{b1}+qk_{b2}=\bar{N}} \left\{ r_1^{p+k_1} r_2^{q+k_2} \right\},$$

donde $k_1 = k_a(N_1 \ominus N)$*,* $k_2 = k_c(N_2 \ominus N)$*.*

La matriz de transferencia de A_2*, teniendo en cuenta los pesos, es* $\begin{pmatrix} k_{b1} & k_{b2} \\ 0 & k_{b2} \end{pmatrix}$*,*

Algoritmo 3.5 Algoritmo para obtener la función *Variant*. Sea $A(\Pi, C, \delta) = (Q, V, q_0, F)$ el autómata correspondiente al lenguaje definido por la aplicabilidad conjunta de las reglas y sea s_j la función generadora del estado q_j, $q_j \in Q$

Requiere: $mset_idx$ // Índice del conjunto que se desea obtener

Devuelve: Conjunto $R \in Appl(\Pi, C, max)$ de tamaño $mset_idx$, considerando que $Appl(\Pi, C, max)$ se encuentra ordenado siguiendo una enumeración inicial fija.

1. $c_state = q_0$ // Estado actual
2. $nb = s_0(n)$ // $NBVariants$
3. $mset = \lambda$ // Conjunto de salida
4. **para** $step = 0$ hasta $mset_idx$ **hacer**
5. Siendo $\{t : (c_state, a_t, q_{j_t})\}$, $1 \leq t \leq k_i$ el conjunto de transiciones de salida del estado c_state, el sistema computa $S(k) = \sum_{m=1}^{k} s_{j_m}(n - step)$. Se establece por definición $S(0) = 0$. Entonces, existe k tal que $S(k) \geq nb$ y no existe $k' < k$ tal que $S(k') > nb$. Sea $t_k : (c_state, a_k, q_{j_k})$ la transición que lo verifica.
6. $c_state = q_{j_k}$
7. $nb = nb - S(k-1)$
8. $mset = out \cdot a_k$
9. **fin para**
10. **devolver** $mset$

y el vector $F = [1, 1]$. Lo que da lugar a la siguiente función generadora para A_2:

$$q_0 = \frac{1}{(1 - x^{k_{b1}})(1 - x^{k_{b2}})}.$$

Los coeficientes $[x^n]q_0$ pueden obtenerse mediante la recurrencia

$$a(n) = a(n - k_{b1}) + a(n - k_{b2}) - a(n - k_{b1} - k_{b2}); n \geq k_{b1} + k_{b2}.$$

Por último, los valores iniciales se establecen según los siguientes casos (se supone que $k_{b1} \geq k_{b2}$):

$$\begin{cases} 1, & n < k_{b1}, \\ 2, & k_{b1} \leq n < k_{b1} + k_{b2} \end{cases}$$

La función *Variant* puede ser calculada según el Algoritmo 3.5. La idea principal de este algoritmo es computar la variante n-ésima utilizando la ordenación

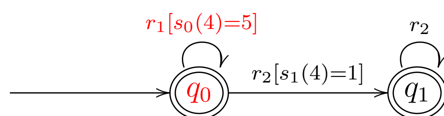
lexicográfica de las transiciones mediante el empleo de un algoritmo similar al del cómputo de un número escrito en un sistema numérico combinatorio. Así, considerando un autómata, obtener una variante de tamaño k a partir de un estado cualquiera, q, se reduce a aplicar sucesivamente k transiciones. En cada transición, $t : (q, r, q')$, se alcanza un estado siguiente, q', al tiempo que se obtiene un elemento, r, de la secuencia de salida. De ese modo, para obtener la variante n-ésima bastará, en cada transición, con ordenar lexicográficamente el conjunto de posibles transiciones, T, y seleccionar la primera transición que verifique que el número de palabras de longitud $k-1$, que pueden obtenerse utilizando todas las transiciones de salida del estado destino, q', es mayor que n.

Ejemplo 5. *Este ejemplo considera como punto de partida el sistema P,* Π_1*, definido en el Ejemplo 3, se obtiene el autómata 3.4, con la función generadora* $q_0 = \frac{1}{(1-x)^2}$*, con* $[x^n]q_0 = n+1$ *y* $[x^n]q_1 = 1$*. Se desea generar el tercer elemento de aquellas cadenas de longitud 5 (*$|w| = 5$*). A continuación se describe la ejecución del algoritmo, acompañada de una representación gráfica, en la que el color verde marca el objetivo, y el color rojo los cambios realizados en el paso del algoritmo al que hace referencia. En color azul se marcan las cadenas descartadas en cada paso.*

Paso 0*. Se parte del estado inicial* q_0*. A continuación se pueden ver los valores del algoritmo, una representación gráfica sobre el autómata y las posibles combinaciones. En este paso,* $s_0(5) = 6$*, donde* $s_0(4) = 5$ *y* $s_1(4) = 1$*. Como* $s_0(4) > 2$*, la transición* (q_0, r_1, q_0) *es la seleccionada, por lo que los nuevos valores del algoritmo son* $c_state = q_0$ *y* $mset = r_1$ *(según la transición seleccionada), mientras que* $nb = s_0(5) - s_1(4) = 5$*, es decir, al valor original de* $nb = NBVariant$ *se le resta la multiplicidad del conjunto descartado, aquel generado desde la transición* (q_0, r_2, q_1)*.*

Paso 0*:*

$$\begin{aligned} c_state &= q_0 &;&\quad c_state = q_0 \\ nb &= s_0(5) = 6 &;&\quad nb = s_0(5) - s_1(4) = 6 - 1 = 5 \\ mset &= \lambda &;&\quad mset = r_1 \end{aligned}$$

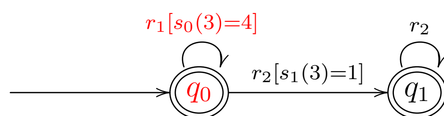


$$Appl(\Pi, C, max) = \left\{ \begin{array}{cccccc} r_1 & r_1 & r_1 & r_1 & r_1 & (0) \\ r_1 & r_1 & r_1 & r_1 & r_2 & (1) \\ r_1 & r_1 & r_1 & r_2 & r_2 & (2) \\ r_1 & r_1 & r_2 & r_2 & r_2 & (3) \\ r_1 & r_2 & r_2 & r_2 & r_2 & (4) \\ r_2 & r_2 & r_2 & r_2 & r_2 & (5) \end{array} \right. \begin{array}{l} \left.\begin{array}{c} \\ \\ \\ \\ \\ \end{array}\right\} q_0(4) \\ \left.\begin{array}{c} \\ \end{array}\right\} q_1(4) \end{array}$$

Paso 1. *Como resultado del anterior paso, el estado actual es* q_0*,* $s_0(4) = 5$*, donde* $s_0(3) = 4$ *y* $s_1(3) = 1$*. Como* $s_0(3) > 2$*, la transición* (q_0, r_1, q_0) *es la seleccionada, por lo que los nuevos valores del algoritmo son* $c_state = q_0$ *y* $mset = r_1 r_1$ *(según la transición seleccionada), mientras que* $nb = nb - s_1(3) = 4$*.*

Paso 1*:*

$$\begin{array}{rll} c_state & = q_0 & ; \quad c_state = q_0 \\ nb & = s_0(4) = 5 & ; \quad nb = s_0(4) - s_1(3) = 5 - 1 = 4 \\ mset & = r_1 & ; \quad mset = r_1 r_1 \end{array}$$

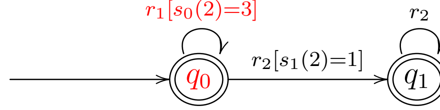


$$Appl(\Pi, C, max) = \left\{ \begin{array}{cccccc} r_1 & r_1 & r_1 & r_1 & r_1 & (0) \\ & r_1 & r_1 & r_1 & r_2 & (1) \\ & r_1 & r_1 & r_2 & r_2 & (2) \\ & r_1 & r_2 & r_2 & r_2 & (3) \\ & r_2 & r_2 & r_2 & r_2 & (4) \end{array} \right. \begin{array}{l} \left.\begin{array}{c} \\ \\ \\ \\ \end{array}\right\} q_0(3) \\ \left.\begin{array}{c} \\ \end{array}\right\} q_1(3) \end{array}$$

Paso 2. *Como resultado del anterior paso, el estado actual es* q_0, $s_0(3) = 4$, *donde* $s_0(2) = 3$ *y* $s_1(2) = 1$. *Como* $s_0(2) > 2$, *la transición* (q_0, r_1, q_0) *es la seleccionada, por lo que los nuevos valores del algoritmo son* $c_state = q_0$ *y* $mset = r_1r_1r_1$ *(según la transición seleccionada), mientras que* $nb = nb - s_1(2) = 4$.

Paso 2:

$$\begin{aligned} c_state &= q_0 & ; \quad c_state &= q_0 \\ nb &= s_0(3) = 4 & ; \quad nb &= s_0(3) - s_1(2) = 4 - 1 = 3 \\ mset &= r_1r_1 & ; \quad mset &= r_1r_1r_1 \end{aligned}$$

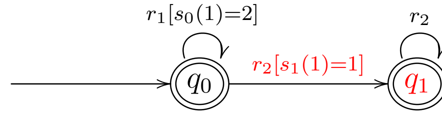


$$Appl(\Pi, C, max) = \left\{ \begin{array}{cccccc} r_1 & r_1 & r_1 & r_1 & r_1 & (0) \\ & & r_1 & r_1 & r_2 & (1) \\ & & r_1 & r_2 & r_2 & (2) \\ & & r_2 & r_2 & r_2 & (3) \end{array} \right. \begin{array}{l} \left.\begin{array}{c} \\ \\ \\ \end{array}\right\} q_0(2) \\ \left.\begin{array}{c} \\ \end{array}\right\} q_1(2) \end{array}$$

Paso 3. *El estado actual es* q_0, $s_0(2) = 3$, *donde* $s_0(1) = 2$ *y* $s_1(1) = 1$. *Como* $s_0(1) \not> 2$, *la transición* (q_0, r_2, q_1) *es la seleccionada, por lo que los nuevos valores del algoritmo son* $c_state = q_1$ *y* $mset = r_1r_1r_1r_2$ *(según la transición seleccionada), mientras que* $nb = nb - s_0(1) = 1$.

Paso 3:

$$\begin{aligned} c_state &= q_0 & ; \quad c_state &= q_1 \\ nb &= s_0(2) = 3 & ; \quad nb &= s_0(2) - s_0(1) = 3 - 2 = 1 \\ mset &= r_1r_1r_1 & ; \quad mset &= r_1r_1r_1r_2 \end{aligned}$$

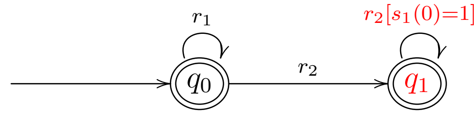


$$Appl(\Pi, C, max) = \left\{ \begin{array}{cccccc} r_1 & r_1 & r_1 & r_1 & r_1 & (0) \\ & & & r_1 & r_2 & (1) \\ & & & r_2 & r_2 & (2) \end{array} \right\} \begin{array}{c} q_0(1) \\ \\ q_1(1) \end{array}$$

Paso 4 y 5. *A partir del paso anterior, el multiconjunto Appl únicamente contiene un elemento. En consecuencia, en este paso y en el siguiente se finaliza la generación del conjunto de reglas de aplicación, R.*

Paso 4 y 5:

$$\begin{array}{rlcrl} c_state & = q_1 & ; & c_state & = q_1 \\ nb & = s_1(1) = 1 & ; & nb & = nb - s_1(0) = 1 - 1 = 0 \\ mset & = r_1r_1r_1r_2 & ; & mset & = r_1r_1r_1r_2r_2 \end{array}$$



$$Appl(\Pi, C, max) = \left\{ \begin{array}{cccccc} r_1 & r_1 & r_1 & r_2 & r_2 & (0) \end{array} \right\} q_1(1)$$

3.3. Sistemas P aceptados por la arquitectura Almond PS

Una vez descritos los fundamentos teóricos, procedimientos y algoritmos en los que se basa la solución propuesta por el doctorando, se detalla qué restricciones impone la arquitectura respecto a los sistemas P de entrada.

En este sentido, tal y como se mencionó en la Sección 2.1, los sistemas P son modelos de computación orientados a máquinas, lo que supone la imposibilidad de

poder diseñar una única máquina capaz de ejecutar cualquier sistema P. Aunque el problema se mantiene en las implementaciones *software*, la mayor flexibilidad que estas ofrecen lo minimiza, existiendo gran cantidad de implementaciones de distintos sistemas P, tal y como se detalla en la Sección 2.2 (Pág. 51). Sin embargo, desde un punto de vista de diseño *hardware*, esta característica supone un reto adicional, ya que implica la generación de una solución *hardware* específica, no únicamente para cada variante de sistema P, sino para cada instancia de cada problema de cada variante. En el caso de los trabajos anteriores de B. Petreska [Petreska and Teuscher, 2004] y de V. Nguyen [Nguyen, 2010], únicamente se ha llegado a implementar una versión determinista del modelo de sistema P transitivo presentado por G. Păun en el artículo fundacional del área [Păun, 2000]. En este sentido, es preciso destacar el algoritmo propuesto por V. Nguyen con el propósito de implementar sistemas P no deterministas, que no tuvo continuación en un diseño y posterior implementación *hardware*, aunque sí fue utilizado como punto de partida para el desarrollo de implementaciones *software* [Cardona et al., 2010; Colomer et al., 2010].

Los fundamentos en los que se basa la arquitectura Almond PS consisten en, para cada transición del sistema, representar el conjunto $Appl(\Pi, C, \delta)$ como un lenguaje no ambiguo y libre de contexto, para posteriormente, empleando series formales de potencias, obtener el número de palabras del lenguaje con la función $NBVariants$ y generar la palabra n-ésima del lenguaje, respetando un orden lexicográfico, con la función $Variant$. En consecuencia, cualquier sistema P para el que en cada transición el conjunto $Appl$ pueda ser representado como un lenguaje libre de contexto y no ambiguo puede ser implementando aplicando los procedimientos detallados anteriormente, lo que supone la práctica totalidad de las variantes de sistemas P estáticos actuales y, probablemente, la gran mayoría de las que se introducirán en un futuro. En este sentido, es preciso destacar que el punto de partida no es la variante de sistema P específica, sino que, como paso previo, cada sistema P de entrada debe ser transformado al entorno formal de redes de células presentado en [Freund and Verlan, 2007], capaz de modelar cualquier sistema P estático, descartando los sistemas P dinámicos. No obstante, este hecho no representa un gran inconveniente, puesto que estos sistemas han de descartarse para ser implementados en *hardware*, debido a las dificultades que supone implemen-

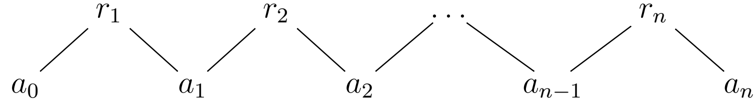
tar un sistema capaz de generar una cantidad de espacio exponencial en tiempo polinomial.

Como puede deducirse del procedimiento formal descrito en la Sección 3.2, la serie formal de potencias empleada para calcular las funciones *NBVariants* y *Variant* depende del lenguaje libre de contexto y no ambiguo que representa el conjunto *Appl*. A su vez, este conjunto viene determinado por las dependencias existentes entre las reglas del sistema, esto es, por el grafo de dependencias de las reglas del sistema P. Por lo tanto, el considerar un tipo de grafo distinto ocasiona cambios en la implementación concreta de las funciones necesarias para elegir, dado un estado inicial, el multiconjunto de reglas que se aplicará para que el sistema evolucione al próximo estado. Evidentemente, esta dependencia es la que afecta a la flexibilidad de la solución propuesta. No obstante, es preciso destacar que la única restricción para que un sistema P sea aceptado se limita al tipo de grafo de dependencia de sus reglas, por lo que el número de sistemas P compatibles con una implementación dada es significativamente mayor que en el caso de las implementaciones previas de B. Petreska y V. Nguyen. Además, los principios de diseño de la arquitectura facilitan la inclusión de nuevos grafos compatibles, al encapsular y aislar los componentes afectados.

Con el propósito de facilitar la comprensión de la arquitectura, se detallará la misma para el sistema P implementado por el doctorando, y posteriormente se detallarán qué componentes pueden verse afectados por la implementación de un nuevo sistema P.

Atendiendo al conjunto de sistemas P implementados, se considera que las reglas de reescritura de los multiconjuntos trabajan en modo maximal, *smax* (Sección 2.1.1). Este modo se corresponde a la ejecución maximal paralela de las reglas, pero en la que estas no pueden aplicarse más de una vez. La definición formal de este modo de derivación puede ser consultada en la definición 7, en la Sección 2.1.1.

A continuación, se considera un sistema de reescritura de multiconjuntos, correspondiente a un sistema P con una membrana, evolucionando en modo *smax*. Para simplificar la construcción se considera que el grafo de dependencia de las reglas tiene forma de cadena sin ponderaciones. A continuación se muestra un grafo genérico.



Sea N_{a_i} el número de objetos a_i en la configuración C, la cantidad de variantes de aplicaciones de una cadena de reglas $r_1, \ldots, r_k$ sobre la configuración C, en modo *smax*, se denota como $NBV([r_1, \ldots, r_k], C)$, $k > 0$. Nótese que para un sistema P, Π, con un conjunto de reglas R, $NBVariants(\Pi, C, smax) = NBV(R, C)$.

Es posible diferenciar tres casos con respecto al número de objetos N_{a_i}, $0 \leq i \leq n$ (considerando que $0 \leq s \leq i \leq e \leq n$):

$N_{a_i} = 0$ Entonces las dos reglas adyacentes, r_i y r_{i+1}, no son aplicables. En este caso las partes de la cadena a la izquierda y a la derecha de a_i son independientes, por lo que el número de variantes es el producto de las variantes correspondientes:

$$NBV(r_s, \ldots, r_e, C) = NBV(r_s, \ldots, r_{i-1}, C) * NBV(r_{i+2}, \ldots, r_e, C)$$

$N_{a_i} > 1$ Al igual que en el caso anterior, la cadena puede dividirse en dos partes ya que ambas reglas, r_i y r_{i+1}, pueden aplicarse:

$$NBV(r_s, \ldots, r_e, C) = NBV(r_s, \ldots, r_i, C) * NBV(r_{i+1}, \ldots, r_e, C)$$

$N_{a_i} = 1$ En este caso no existen recursos suficientes para que ambas reglas, r_i y r_{i+1}, puedan aplicarse, existiendo una competición entre ambas por el recurso común, a_i.

La última situación es la que presenta especial interés desde un punto de vista de algoritmia e implementación. Sin perder generalidad, es posible suponer que $N_{a_i} = 1, 0 \leq i \leq n$. Nótese que el lenguaje de cadenas binarias de longitud n correspondientes al vector conjunto de aplicabilidad de reglas $r_1, \ldots, r_n$, coincide con el lenguaje L_I del Ejemplo 2. Por tanto, la cantidad de posibilidades de aplicación de

este tipo de cadena de reglas de longitud n es igual a $NBV(r_1, \ldots, r_n, C) = [x^n]q_0$, *i.e.* , $q_0(0) = 1$, $q_0(1) = 1$, $q_0(2) = 2$ y $q_0(n) = q_0(n-2) + q_0(n-3)$, $n > 3$.

Así, con objeto de hallar $NBVariants(\Pi, C, smax)$, es necesario dividir la cadena en uno o más fragmentos de longitud variable, en función de las multiplicidades de los objetos, para posteriormente aplicar la función NBV a cada fragmento, según la descomposición anterior.

La función *Variant* para cada fragmento puede obtenerse utilizando el Algoritmo 3.5.

3.4. Implementación *hardware*

A continuación se detalla la implementación *hardware* de la arquitectura Almond PS, una vez presentados los fundamentos empleados para su diseño.

Con el objetivo de obtener una arquitectura con la mayor escalabilidad y flexibilidad posible, se ha optado por emplear un enfoque modular, manteniendo un nivel de abstracción uniforme, el cual permite, adicionalmente y en la medida de lo posible, favorecer el rendimiento global del sistema. De ese modo, el diseño se estructura en una serie de módulos con interfaces claramente definidas. Cada una de las tareas principales del algoritmo recae sobre cada uno de ellos, existiendo comunicación únicamente con los adyacentes: módulo anterior (cuyas salidas son sus entradas) y posterior (que recibe sus salidas).

El grafo de dependencias presentado en la sección anterior se ha tomado como punto de partida para el modelado de los sistemas P. Tal y como se ha comentado anteriormente, esta aproximación reduce la complejidad al eliminar elementos estructurales redundantes como cargas, tipos de objetos y relaciones entre membranas, entre otros, simplificándolos a relaciones de dependencia entre reglas y objetos, con una codificación adecuada. Además, este enfoque simplifica la división de tareas entre los distintos módulos, facilitando lograr un grado de encapsulación suficiente para alcanzar los objetivos de diseño propuestos. Atendiendo a la representación de los componentes estructurales del modelo, los objetos se representan explícitamente mediante registros. Sin embargo, no existe una correspondencia directa entre regla y módulo *hardware* que implemente su funcionalidad. En este sentido, la lógica de las reglas se ha distribuido transversalmente entre los dis-

tintos componentes que forman la arquitectura Almond PS, según los algoritmos descritos anteriormente.

Partiendo de la división inicial en fases de la ejecución de un sistema P, realizada en la Sección 3.1.1, se ha efectuado la siguiente división de tareas:

1. *Persistencia*: obviamente, es necesario almacenar los resultados de cada transición, así como el resultado final.

2. *Cálculo de máxima aplicabilidad para cada regla, o* N_{r_i}: consiste en obtener el número máximo de veces que puede aplicarse una regla, sin considerar las relaciones de dependencia que pudieran existir con el resto de reglas del modelo. Aunque la forma en la que se construye *Appl* y en la que se selecciona el multiconjunto de reglas depende del tipo de sistema P, esta operación auxiliar es independiente de este cómputo y, a la vez, común a la mayoría de los algoritmos de ejecución de sistemas P (independientemente de su tipo).

3. *Asignación de aplicabilidad, o* n_{r_i}: comprende seleccionar qué reglas se aplicarán, y cuántas veces cada una de ellas. Considerando la ejecución teórica, se corresponde con seleccionar el multiconjunto de reglas aplicables, R, del conjunto *Appl*.

4. *Aplicación de reglas*: conlleva calcular los cambios que ocasionará en la configuración actual, C_i, la ejecución de las reglas según el multiconjunto R generado en la anterior tarea, obteniendo, de ese modo, la siguiente configuración, C_{i+1}.

5. *Actualización de configuración actual*: una vez calculados los cambios en las multiplicidades de los objetos, es necesario actualizar la configuración actual convenientemente.

6. *Detección de condición de parada*: finalmente, el sistema debe verificar si se ha alcanzado una condición de parada.

7. *Control del sistema*: recoge la funcionalidad de control.

En base a esta división de funcionalidad, la arquitectura Almond PS se divide en seis bloques: bloque de entrada/salida, bloque de persistencia, bloque de cómputo de N_{r_i}, bloque de asignación, bloque de aplicación y bloque de control. De estos, dos se dedican a tareas genéricas, no específicas de la ejecución de sistemas P: el dedicado al control del sistema (bloque de control), y el dedicado a la interfaz de entrada/salida (bloque de entrada/salida). Así, toda la funcionalidad específica de los sistemas P se consigue mediante los cuatro bloques restantes, los cuales respetan los principios de diseño establecidos anteriormente. Todos ellos requieren únicamente de un solo ciclo de reloj para llevar a cabo sus tareas, excepto el bloque de asignación, que precisa de dos ciclos. En consecuencia, el sistema desarrollado es capaz de calcular y aplicar una transición y almacenar la nueva configuración en tan sólo cinco ciclos de reloj.

En la Fig. 3.2 se presenta una visión general de la arquitectura Almond PS, en la que se relaciona la correspondencia entre la división de tareas inicial (3.1.1) y la establecida en este punto, además de su distribución entre los distintos bloques que componen la arquitectura Almond PS.

3.4.1. Bloque de persistencia

Este bloque implementa la persistencia, la actualización de los valores de multiplicidad de los objetos y algunos aspectos de la fase de parada. En cuanto a las dos primeras, son independientes del tipo del sistema, dependiendo únicamente de la cantidad de objetos distintos y de sus multiplicidades máximas. En este sentido, es preciso destacar que, aunque el entorno formal de redes de células emplee una codificación para reducir los elementos estructurales del sistema P, esto no implica forzosamente un número mayor de objetos del sistema a almacenar. Este hecho es obvio si se considera un sistema P, Π_2, con dos membranas, m_1 y m_2, y un tipo de objeto, a. Al considerar Π_3, obtenido de aplicar el entorno formal sobre Π_2, el objeto a se codifica como a_1 (objeto a presente en m_1), y objeto a_2 (objeto a presente en m_2). Desde un punto de vista teórico, en Π_3 se ha doblado el número de objetos presentes en el sistema Π_2, aunque desde un punto de vista práctico, en Π_2 es necesario almacenar los objetos a presentes en cada membrana, siendo necesario mantener dos variables y, en conclusión, el mismo número de objetos presentes en

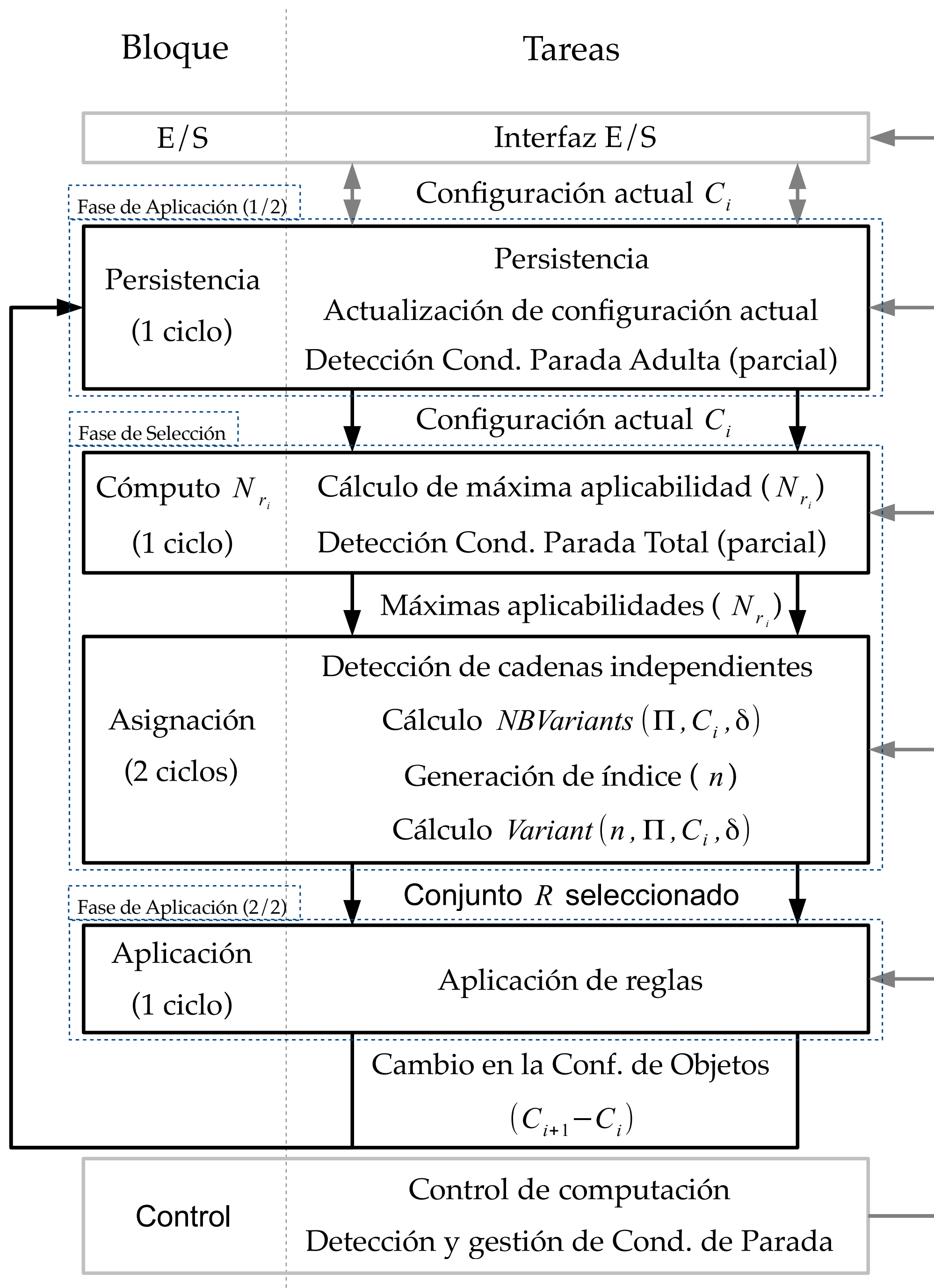


Fig. 3.2: Visión general de la arquitectura Almond PS. La figura muestra tanto los bloques principales como el flujo de información entre ellos. En líneas punteadas se detalla la relación entre la división de tareas establecida en la Sección 3.1.1 y la de esta figura.

Π_3. Es fácil aplicar este razonamiento a los objetos consumidos y producidos por las reglas, así como a cualquier elemento adicional introducido en el modelo, como las cargas asociadas a membranas.

Atendiendo a la implementación *hardware*, cada objeto se corresponde con un registro que almacena su multiplicidad en la configuración actual como un entero sin signo. Esto permite ahorrar recursos *hardware*, ya que no es posible contar con un número negativo de objetos en un sistema P. No obstante, también contempla el almacenamiento de un número con signo en *Ca2*.

Cada registro recibe como entrada los cambios de multiplicidad del objeto para la próxima configuración del sistema. Así, un sumador se encarga de la operación de actualización de la configuración actual. En este sentido, como la multiplicidad puede disminuir, este componente aritmético sí emplea enteros con signo en *Ca2*, llevando a cabo la conversión del valor almacenado en el registro de entero sin signo a entero con signo. Debido a que la multiplicidad del objeto es siempre positiva, las conversiones entre valor almacenado y sumando, resultado de la suma y valor de entrada para el registro, consisten en añadir y descartar el bit más significativo (un 0), lo que no supone un impacto negativo en el consumo de recursos. En caso de que se opte por almacenar el resultado como entero con signo, estas conversiones no son implementadas.

Adicionalmente, este módulo implementa parte de la funcionalidad del cómputo de la condición de parada adulta. Concretamente, la detección de aquella configuración C_i, tal que $C_j = C_i$, $\forall j \geq i$. Aunque esta decisión de diseño puede comprometer el principio de modularidad del sistema al distribuir una funcionalidad entre varios módulos, es necesaria para mantener la encapsulación de los distintos componentes del sistema. De ese modo, el bloque recoge únicamente la lógica asociada a la detección de la condición, ofreciendo un servicio de detección. Es responsabilidad del bloque de control actuar cuando se active esta condición. Su implementación consiste en un comparador por cada registro comparando el nuevo valor y el almacenado en la configuración anterior, combinado con un contador, y junto con una puerta lógica *AND* que recibe la salida de los comparadores de los registros. Debido a la complejidad que supondría asegurar la condición de parada adulta, se realiza una aproximación, en la que se acota el mínimo número de transiciones necesario para que se considere que el sistema ha alcanzado una confi-

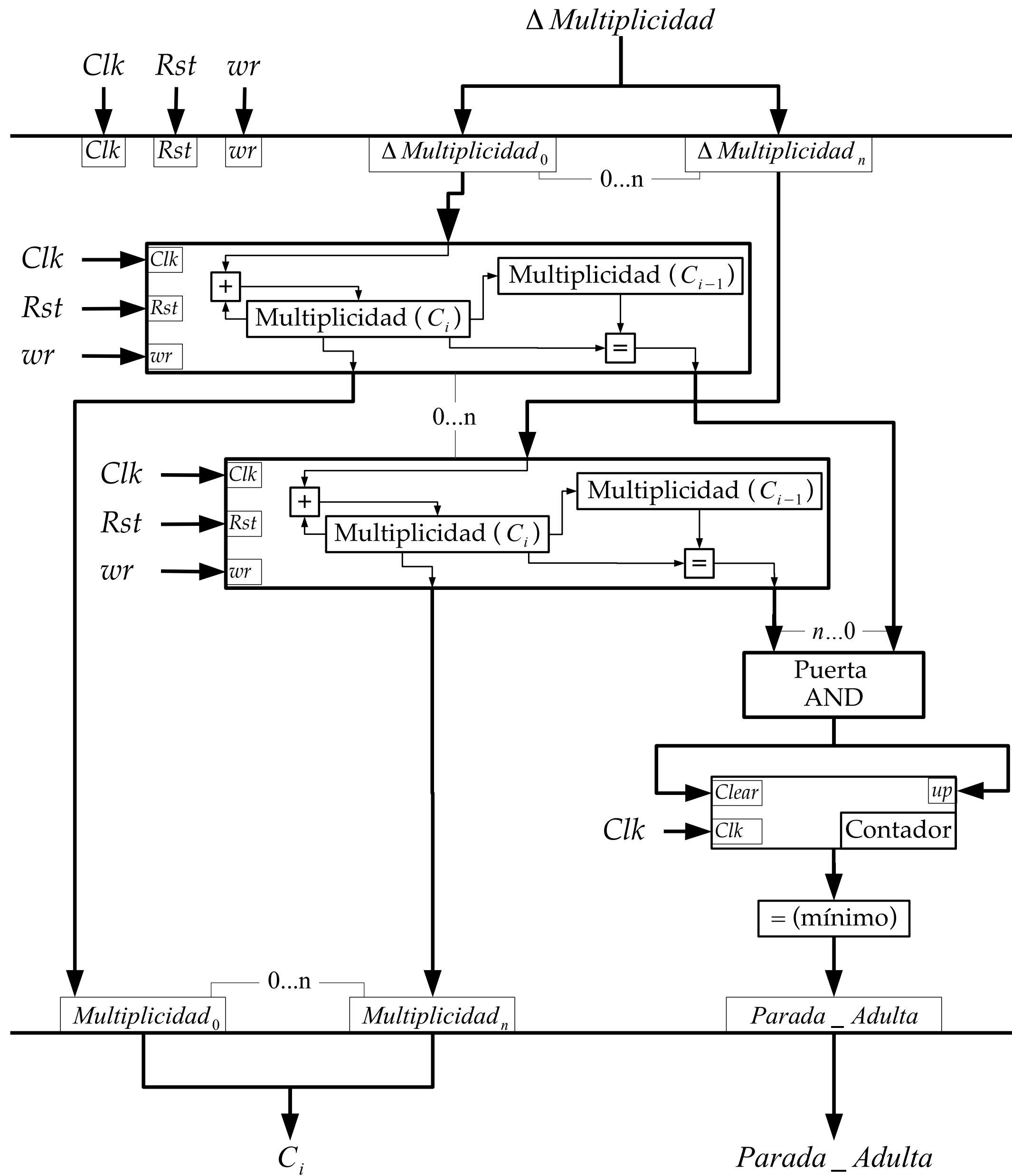


Fig. 3.3: Esquemático del bloque de persistencia. Se visualizan los detalles más significativos.

.

guración de parada adulta. Este número es configurable por el usuario, siendo dos su valor por defecto, lo que significa que la condición de parada adulta es detectada cuando tres transiciones consecutivas generan tres configuraciones iguales.

Respecto a las líneas de salida, ofrece la configuración actual al bloque de cómputo de N_{r_i} y la señal que indica la condición de parada al bloque de control.

3.4.2. Bloque de cómputo de N_{r_i}

Se caracteriza por ser un bloque eminentemente combinacional. Recibe como entrada la multiplicidad de los objetos en la configuración actual y calcula N_{r_i}, o el número máximo de aplicabilidad de cada regla, independientemente del resto.

Esta operación es independiente del modo de derivación del sistema. Sin embargo, el hecho de considerarlo puede ayudar a optimizar el diseño en área y rendimiento. Por ello, se ofrece la alternativa de implementar este bloque a medida para el modo de derivación seleccionado. En los sistemas P aceptados por la arquitectura Almond PS (Sección 3.3), el modo de derivación es *smax* (Sección 2.1.1), lo que determina que no es necesario hacer el cómputo global, basta con comprobar que existen suficientes recursos para que cada regla pueda ser ejecutada al menos una vez.

Si se opta por una implementación acoplada al modo de derivación, Fig. 3.4(1), puede ser necesario calcular otros parámetros. En este caso y, generalizando para cualquier modo de derivación $smax_j$, es necesario distinguir entre aquellos casos en los que haya suficientes recursos para que las reglas puedan ser ejecutadas un número j de veces, aquellos en los que no existen recursos suficientes para que ninguna regla pueda ser ejecutada, y aquellos en los que los recursos permiten la ejecución únicamente de algunas de las reglas, no todas, existiendo competición por los recursos. Es por ello que es en este bloque donde se detectan los extremos de las cadenas de reglas independientes (Sección 3.3), formadas por secuencias de reglas dependientes por alguno de los recursos que consumen. Respecto a las líneas de salida, por cada regla se ofrecen dos líneas: una de ellas, que indica la multiplicidad de la regla en el modo *smax*, 0 ó 1, y otra línea que determina cuándo la regla es dependiente de sus adyacentes debido a la existencia de recursos suficientes o a la falta de estos, es decir, aquellos casos en los que $N_{r_i} = 0$.

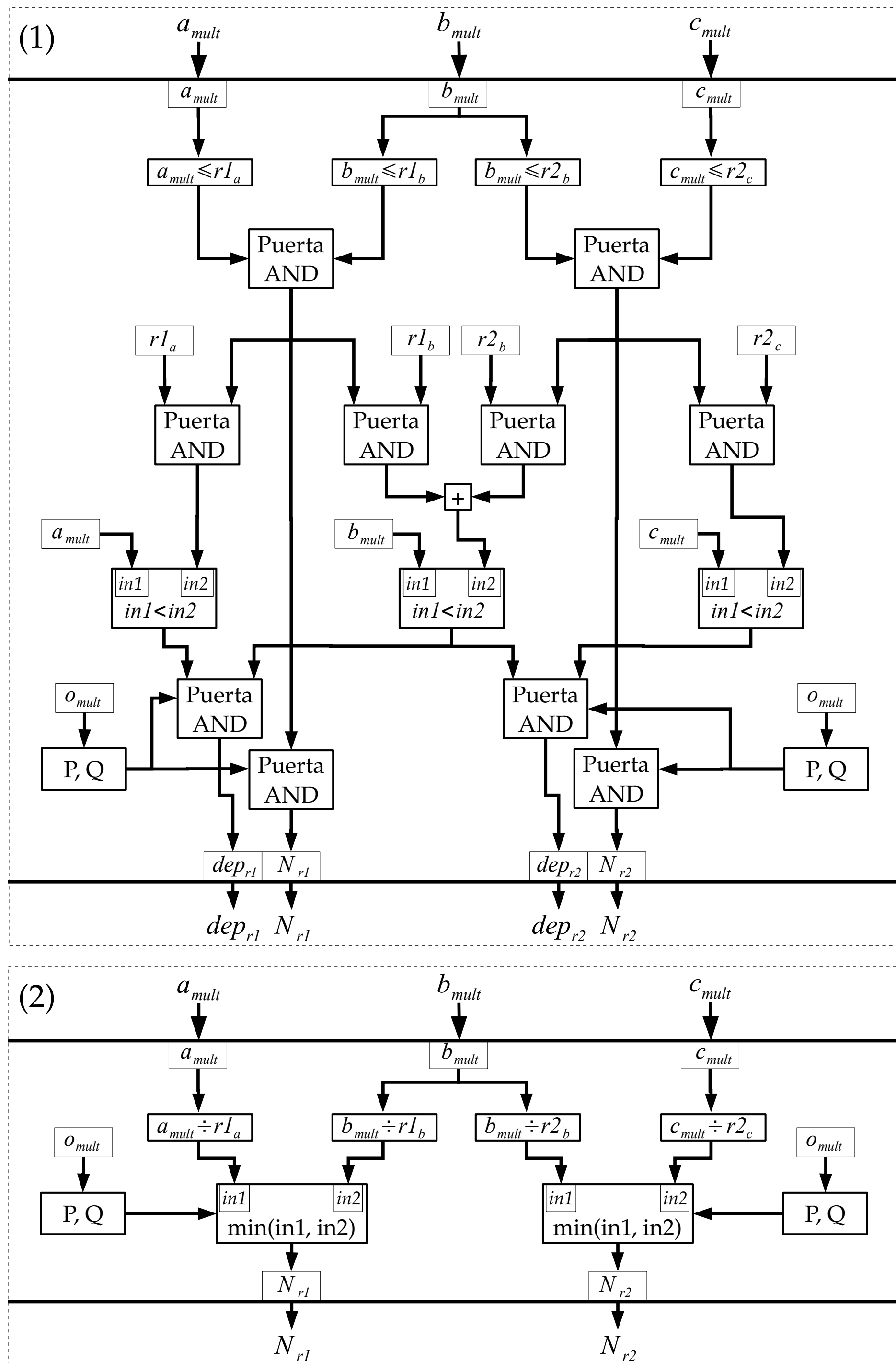


Fig. 3.4: Esquema del bloque de cómputo de N_{r_i}. Para facilitar la comprensión, se ha preferido utilizar un ejemplo, en el que $R = \{r1, r2\}$ con $lhs(r1) = ab$ y $lhs(r2) = bc$. En (1) se presenta la implementación empleada para el modo acoplado con $smax$. En (2) se muestra una implementación genérica, independiente del modo de transición.

En la implementación no acoplada es necesario emplear divisores por cada uno de los objetos consumidos por cada regla, Fig. 3.4(2). En caso de que los divisores sean exclusivos de cada par (regla, objeto), es posible reducir la lógica del módulo empleando divisores con divisor constante. No obstante, en una implementación a medida su uso se encuentra condicionado por el diseño del módulo, pudiendo ser sustituidos por comparadores en ocasiones. Por ejemplo, cuando el *hardware* esté optimizado para un modo de derivación $smax_j$, será necesario añadir comparadores que determinen cuando $N_{r_i} = 0$ y $N_{r_i} \geq j$. En ambos casos, una vez que se ha obtenido el número de veces que puede ser aplicada una regla en base a uno de los objetos de su parte izquierda, es necesario obtener el mínimo de entre todos aquellos que pertenecen a la parte izquierda de la regla, obteniendo de ese modo N_{r_i}.

Independientemente del modo de generación, un aspecto común es la comprobación de los conjuntos de reglas P y Q. Estos elementos pueden impedir de forma incondicional la posible ejecución de una regla, en el caso que algún elemento del conjunto P no exista en la configuración actual, o que exista algún elemento del conjunto Q. Así, su implementación consiste en el chequeo de las condiciones mediante el empleo de puertas lógicas que, finalmente, modifican, en caso de ser necesario, las distintas líneas de salida del bloque. Para el caso de la generación no acoplada, es posible incorporar la salida del chequeo de las condiciones P y Q a la entrada del comparador, aglutinando la lógica en el componente.

3.4.3. Bloque de asignación

El bloque de asignación es el más destacado de la arquitectura Almond PS, ya que es el encargado de generar el conjunto de aplicabilidad de reglas, R, que supone el mayor desafío a la hora de ejecutar este tipo de sistemas. Este bloque se encuentra acoplado al modo de derivación del sistema P, por lo que es necesario modificarlo para distintos modos. En este caso, el modo empleado es el $smax$, en un sistema P de la clase presentada anteriormente. En la Figura 3.5 se detalla a nivel estructural este bloque. A lo largo de esta sección, especialmente en los algoritmos presentados, se empleará la notación empleada en esta.

Atendiendo a la funcionalidad del módulo, en el Algoritmo 3.6 se presenta a

alto nivel la algoritmia recogida en este bloque. Recibe como entrada el número máximo de aplicabilidad de cada regla, según el modo de derivación, junto con una señal indicando si la ejecución de la regla es independiente de sus adyacentes en esa transición. Aplicando los métodos y el Algoritmo 3.5 detallados en 3.2, el bloque genera y ofrece como salida al bloque de aplicación el conjunto R.

Respecto a la implementación *hardware*, es preciso destacar que el cómputo de $NBVariants(\Pi, C, smax)$ utiliza un subconjunto de operaciones necesarias para calcular $Variant(R_idx, \Pi, smax)$. Del mismo modo, aunque la generación del número aleatorio R_idx depende del valor de la función $NBVariants(\Pi, C, smax)$, es posible generar ambos simultáneamente. Considerando a_n los coeficientes de la serie de potencias asociada al autómata que describe el lenguaje asociado (Sección 3.1.3 y Ejemplo 3), cada a_n se obtiene de la suma de los coeficientes a_{n-2} y a_{n-3}. Por lo tanto, el mínimo número de bits necesarios para representar el coeficiente a_n en binario es $1 + log_2(max(a_{n-2}, a_{n-3}))$. Además, $a_0 = a_1 = 1$, y $a_2 = 2$, verificándose que $a_n < 2^n$ para cualquier coeficiente de la serie de potencias. Por lo tanto, para representar en binario cada coeficiente basta incrementar en un bit el ancho del número de cada pareja de coeficientes, empezando con un único bit. Para generar un número (rn) válido ($0 \leq rn < NBVariants$), es preciso el uso de máscaras que adecúen el valor máximo, junto con una corrección para el caso en que $rn \geq NBVariants$, es decir, aquellos casos en los que $NBVariants$ no sea múltiplo de 2.

Considerando las dependencias entre las operaciones, el cómputo global requiere de tres pasos: hallar $NBVariants$, generar de forma no determinista el índice i, tal que $0 \leq i < NBVariants$, y generar el elemento i-ésimo. Todas estas operaciones son dependientes y, en consecuencia, bloqueantes. No obstante, anteriormente se ha demostrado que el índice y $NBVariants$ pueden ser calculados simultáneamente, reduciéndose el cálculo a dos pasos. En la implementación propuesta, cada paso se completa en un ciclo de reloj, por lo que son necesarios únicamente dos ciclos de reloj para que, partiendo de las aplicabilidades máximas de las reglas, N_{r_j}, se genere un multiconjunto de aplicabilidad $\mathcal{R}$.

Atendiendo a la arquitectura, el bloque se descompone en unidades funcionales mínimas ($RuleBlock_{r_j}$), con la lógica de interconexión adecuada (Fig. 3.5). Atendiendo al sistema P, cada unidad funcional representa una regla del sistema

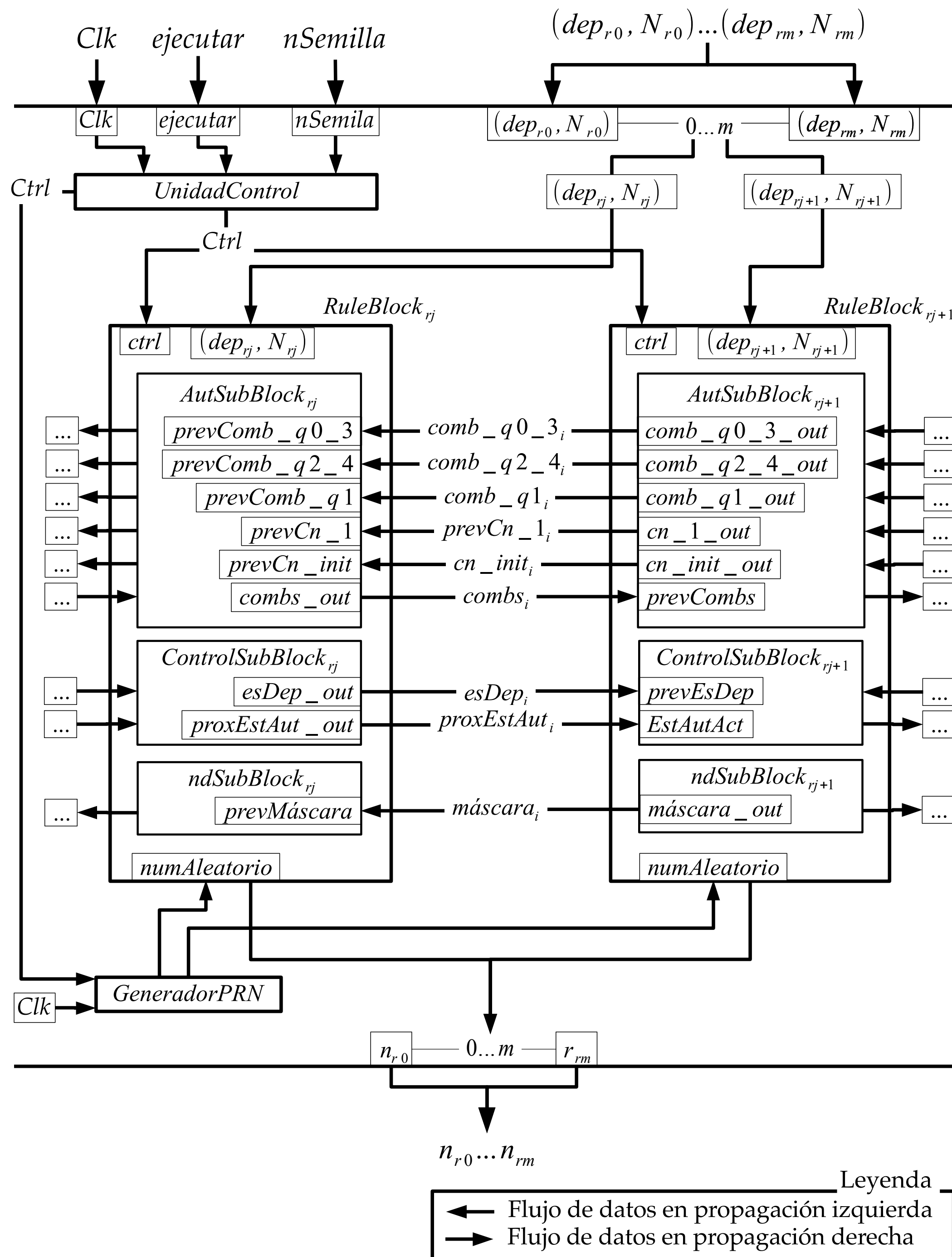


Fig. 3.5: Detalle de la arquitectura del bloque de asignación. Se muestra una generalización de los componentes del bloque: entrada, salida y subbloques principales, así como el principal flujo de datos interno. Para una mejor comprensión del flujo de datos, se detallan dos bloques genéricos consecutivos. En este sentido, la flechas hacia la izquierda indican el flujo de datos durante la propagación izquierda, y las flechas hacia la derecha el flujo de información en la propagación derecha.

Algoritmo 3.6 Algoritmo que detalla en alto nivel el conjunto de tareas implementadas por el módulo *hardware*.

Requiere: N_j: aplicabilidad máxima de cada regla.
Requiere: *dep*: variable que indica si la ejecución de la regla es independiente del resto de variables del sistema P.
Requiere: *rn_retries*: indica el número de intentos para obtener un número aleatorio válido.
Devuelve: $\mathcal{R}$: Conjunto de aplicabilidad de reglas.

1. $\mathcal{R} = \emptyset$
2. Cómputo del conjunto $frag = \{fr_j\}$, $(0 \leq j < k)$. Es generado dividiendo la cadena de entrada en k secuencias independientes, tal como se describe en la Sección 3.3.
3. **para** fr_j en $frag$ **hacer**
4. El sistema obtiene la función $NBVariants(\Pi, C, smax)$ aplicada sobre la secuencia fr_j. Para ello utiliza los procedimientos comentados en las secciones 3.2 y 3.3.
5. $cur_rn_retries = rn_retries - 1$
6. Se genera un número, rn_j, empleando un método no determinista y dependiente del sistema P. Para el sistema P actual se calcula un número aleatorio empleando un generador *Linear Feedback Shift Register* (LFSR).
7. **mientras** $(cur_rn_retries > 0)$ y $(rn_j \notin [0, NBVariants_j(\Pi, C, smax)))$ **hacer**
8. $cur_rn_retries - = 1$
9. Se genera un número, rn_j, empleando un método no determinista y dependiente del sistema P. Para el sistema P actual se calcula un número aleatorio empleando un generador LFSR.
10. **fin mientras**
11. **si** $rn_j \notin [0, NBVariants_j(\Pi, C, smax))$ **entonces**
12. $R_idx_j = rn_j + NBVariants_j(\Pi, C, smax))$
13. **sino**
14. $R_idx_j = rn_j$
15. **fin si**
16. $R_j = Variant_j(R_idx_j, \Pi, C, smax)$. Para ello se aplica la función *Variant* sobre la secuencia fr_j, según el Algoritmo 3.5.
17. $\mathcal{R} = \mathcal{R} \bigcup R_j$
18. **fin para**
19. **devolver** $\mathcal{R}$

Algoritmo 3.7 Algoritmo implementado en el bloque de asignación. Casi la totalidad de las operaciones son realizadas a nivel local en cada bloque (Fig. 3.5), durante la propagación izquierda (Algoritmos 3.8 y 3.9) y la propagación derecha (Algoritmos 3.10 y 3.11). Cada boque **para todo** es ejecutado en un único ciclo de reloj.

Requiere: $\{(dep_{rj}, N_{rj})\}$: Conjunto ordenado de pares que indica si la ejecución de la regla es independiente del resto del sistema P, dep_{rj}, y la aplicabilidad máxima de cada regla, N_{rj}.
Devuelve: $\{n_{rj}\}$: Conjunto ordenado de aplicabilidad de las reglas del sistema P.

1. **para todo** $RuleBlock_{r_j}$ **hacer**
2. propagación_izquierda() // Algoritmo 3.8
3. **fin para**
4. **para todo** $RuleBlock_{r_j}$ **hacer**
5. propagación_derecha() // Algoritmo 3.10
6. **fin para**
7. **devolver** $\{n_{rj}\}$

y, respecto al autómata asociado, un estado. Manteniendo el principio de encapsulación del diseño, cada unidad funcional está conectada únicamente a una unidad anterior y a otra posterior. Cada unidad se descompone en tres subbloques: $AutSubBlock_{rj}$ encargado de la generación de las funciones asociadas al autómata ($NBVariants$ y $Variant$), $ndSubBlock_{rj}$ asociado a la generación del número aleatorio, y $ControlSubBlock_{rj}$ que implementa la lógica de control en el cómputo de las operaciones. Adicionalmente, se requiere un módulo encargado de la generación de números aleatorios en bruto.

Funcionalmente, se mantiene la unidad mínima $RuleBlock_{r_j}$, tal y como se describe en el Algoritmo 3.7. Las operaciones se estructuran en dos grupos: propagación izquierda y propagación derecha, siguiendo un modo de cómputo y transmisión de resultados similar a una topología *daisy chain*. Tal como se muestra en el Algoritmo 3.7, la propagación izquierda (Algoritmo 3.8) es la primera en ejecutarse. En esta sub-fase se ejecutan los pasos $1-5$ del Algoritmo 3.6 (generación del número aleatorio y $NBVariants$), con un flujo de datos desde la última regla hasta la primera. Para el cálculo de $NBVariants$, y considerando la función generadora asociada, $a_n = a_{n-2} + a_{n-3}$ con valores constantes para los primeros 3 elementos, es preciso transmitir los valores asociados a cada estado del autómata: q_0, q_1, q_2,

q_3, q_4 y el valor q_0 y q_3 de la anterior unidad. Se puede comprobar que los valores de la función generadora para los estados q_0 y q_3, y q_2 y q_4 son equivalentes, por lo que se reduce la transmisión a cuatro valores: q_{0_3}, q_1, q_{2_4} y $Prev_q_{0_3}$. Adicionalmente, es preciso transmitir una variable, Cn_init, que indique cuándo las unidades representan alguno de los tres primeros coeficientes (Algoritmo 3.9). Para este último caso, es preciso almacenar los valores de los tres primeros coeficientes en cada bloque $RuleBlock_{r_j}$. Atendiendo a la generación del número aleatorio, en primer lugar se genera un número con el ancho de bits necesario para ofrecer suficiente precisión en cualquier situación, por lo que debe contemplarse aquella situación en la que todas las reglas formen parte de una única secuencia ($k = 1$). Este número es calculado por un sub-módulo que implementa un generador *Linear Feedback Shift Register* (LFSR), y las operaciones necesarias son solapadas por las de los otros módulos de la arquitectura Almond PS, ya que el valor es independiente de la configuración en la que se encuentre el sistema P. A continuación, la lógica implementada en las unidades $RuleBlock_{r_j}$ selecciona el fragmento de bits necesario para representar la salida de la función $NBVariants$.

En la propagación derecha (Algoritmo 3.10) se procesan los pasos $6 - 14$ del Algoritmo 3.6 ($Variant$), en un sentido de izquierda a derecha. Para ello, basta con aplicar los pasos del Algoritmo 3.5 y hacer uso de los valores intermedios calculados para $NBVariants$ y el número aleatorio obtenido en la propagación izquierda. De ese modo, en función del estado en el que se encuentre el autómata representado por el módulo, se generará el número de aplicabilidad para cada regla (Algoritmo 3.11), de forma no determinista y siguiendo una distribución pseudo-multiprobable. Respecto al número aleatorio, rn, es preciso contemplar aquellos casos en los que $rn \geq NBVariants$, ya que en ellos se debe aplicar la corrección pertinente. En este sentido, se contemplan 2 posibles escenarios. El primero de ellos es generar números aleatorios hasta obtener uno, rn, que verifique $rn < NBVariants$. Esta solución prima la corrección de la solución frente al rendimiento. No obstante, el sistema es capaz de generar 4 números aleatorios en segundo plano, mientras que se ejecutan el resto de pasos de la transición. Es por ello que se reduce el impacto en el rendimiento, ya que existe un $6,25\,\%$ de probabilidad de que sea preciso generar más números aleatorios, retrasando así la ejecución del sistema. La segunda opción persigue un compromiso entre

Algoritmo 3.8 - propagación_izquierda. Algoritmo que detalla las operaciones que tienen lugar durante la propagación izquierda en cada bloque. Cada bloque representa una regla e implementa el autómata asociado al tipo de sistema P aceptado. Así, el autómata (bloque) se encuentra en un estado distinto en cada computación. La Fig. 3.5 muestra los valores propagados entre los distintos bloques. Nótese que es el autómata el que determina las operaciones a realizar en esta etapa, el algoritmo presente es válido para el autómata descrito en el Ejemplo 3

Requiere: (dep_r, N_r): Tupla que indica si la ejecución de la regla es independiente del resto del sistema P, dep, y su aplicabilidad máxima, N_r.

Requiere: Combinaciones asociadas a los estados $Q0_ST$ y $Q3_ST$ ($prevComb_q0_3$), $Q2_ST$ y $Q4_ST$ ($prevComb_q2_4$) y $Q1_ST$ ($prevComb_q1$) del bloque anterior.

Requiere: $prevCn_init$: Contador que indica si en el anterior bloque se generó alguno de los 3 primeros coeficientes de la secuencia de Padovan.

Requiere: $prevCn_1$: Combinaciones posibles desde el estado en que se encuentra el bloque anterior.

Requiere: $prevMáscara$: Valor acumulado de la máscara del bloque anterior.

Devuelve: Combinaciones asociadas a los estados $Q0_ST$ y $Q3_ST$ ($comb_q0_3_out$), $Q2_ST$ y $Q4_ST$ ($comb_q2_4_out$) y $Q1_ST$ ($comb_q1_out$) del bloque actual.

Devuelve: cn_init_out Contador que indica si en el bloque actual se ha generado alguno de los 3 primeros coeficientes de la secuencia de Padovan.

Devuelve: cn_1_out: Combinaciones posibles desde el estado en que se encuentra el bloque actual.

Devuelve: $máscara_out$: Valor acumulado de la máscara.

1. $valores_aux_padovan()$ // Algoritmo 3.9
2. **si** (dep_r, N_r) indica que la regla es independiente **entonces**
3. $cn_init_out = 0$
4. $máscara_out = 0$
5. **sino**
6.   **si** $prevCn_init \neq 3$ **entonces**
7. $cn_init_out += 1$ // Primeros 3 coeficientes de Padovan
8.   **sino**
9. $cn_init_out = prevCn_init$
10.   **fin si**
11.   **si** $(prevMáscara + 1) < prevComb_q0_3$ **entonces**
12. $máscara_out = (prevMáscara << 1)\ \&\ 0x1$
13.   **sino**
14. $mscara_out = prevMáscara$
15.   **fin si**
16. **fin si**

Algoritmo 3.9 - valores_aux_padovan. Algoritmo que detalla las operaciones auxiliares necesarias para generar el coeficiente actual, así como los posteriores, de la secuencia de Padovan. Por motivos de espacio únicamente se muestran los nombres de los parámetros de entrada y salida. Su descripción puede ser consultada en el Algoritmo 3.8

Requiere: $prevCn_init$; $prevComb_q0_3$; $prevCn_1$; $prevComb_q1$

Devuelve: $comb_q0_3_out$; cn_1_out; $comb_q1_out$; $comb_q2_4_out$. Valores auxiliares necesarios para generar el coeficiente de la secuencia de Padovan asociado al bloque actual y posteriores.

1. **si** $prevCn_init == 0$ **entonces**
2. $comb_q0_3_out = 1$
3. $cn_1_out = 1$
4. $comb_q1_out = 1$
5. $comb_q2_4_out = 1$
6. **sino si** $prevCn_init == 1$ **entonces**
7. $comb_q0_3_out = 2$
8. $cn_1_out = prevComb_q0_3$
9. $comb_q1_out = 1$
10. $comb_q2_4_out = 1$
11. **sino**
12. $comb_q0_3_out = prevCn_1 + prevComb_q1$
13. $cn_1_out = prevComb_q0_3$
14. $comb_q1_out = prevCn_1$
15. $comb_q2_4_out = prevComb_q1$
16. **fin si**

Algoritmo 3.10 - propagación_derecha. Algoritmo que detalla las operaciones que tienen lugar durante la propagación derecha en cada bloque (Fig. 3.5). Cada bloque representa una regla e implementa el autómata asociado al tipo de sistema P aceptado. Así, el autómata (bloque) se encuentra en un estado distinto en cada computación. La Fig. 3.5 muestra los valores propagados entre los distintos bloques.

Requiere: (dep_r, N_r): Par que indica si la ejecución de la regla es independiente del resto del sistema P, *dep*, y su aplicabilidad máxima, N_r.
Requiere: $comb_q0_3_out$ Combinaciones asociadas a los estados $Q0_ST$ y $Q3_ST$ del bloque actual.
Requiere: $prevComb_q1$ Combinaciones asociadas al estado $Q1_ST$ del bloque anterior.
Requiere: *numAleatorio* Número generado por el LFSR.
Requiere: $máscara_out$ Valor acumulado de la máscara.
Requiere: *estAutAct* Estado al que dirige la transición seleccionada en el bloque anterior, por lo que representa el estado en que se encuentra el bloque actual.
Requiere: *prevCombs* Número de posibles combinaciones del bloque anterior.
Requiere: *prevEsDep* Indica si el anterior bloque es independiente al actual.
Devuelve: $combs_out$ Número de posibles combinaciones del bloque actual.
Devuelve: $proxEstAut_out$ Estado al que dirige la transición seleccionada en el bloque actual.
Devuelve: $esDep_out$ Indica si el bloque actual es independiente del posterior.
Devuelve: n_r Número de aplicaciones de la regla que representa el bloque.

1. **si** (dep_r, N_r) indica que la regla es independiente **entonces**
2. $n_r = N_r$
3. $esDep_out = 0$
4. **sino**
5. $esDep_out = 1$
6. **si** $prevEsDep == 1$ **entonces**
7. $estAut_var = estAutAct$
8. $combs_var = prevCombs$
9. **sino**
10. $estAut_var = Q0_ST$
11. $combs_var = numAleatorio \ \& \ máscara_out$
12. **si** $combs_var \geq comb_q0_3_out$ **entonces**
13. $combs_var+ = prevComb_q0_3$
14. **fin si**
15. **fin si**
16. $asignación_dependiente()$ // Algoritmo 3.11
17. **fin si**
18. **devolver** n_r

Algoritmo 3.11 - asignación_dependiente. Algoritmo que detalla las operaciones auxiliares que tienen lugar durante la propagación derecha en cada bloque. Cada bloque representa una regla e implementa el autómata asociado al tipo de sistema P aceptado. Así, el autómata (bloque) se encuentra en un estado distinto en cada computación. Por motivos de espacio únicamente se muestran los nombres de los parámetros de entrada y salida. Su descripción puede ser consultada en el Algoritmo 3.10.

Requiere: $estAut_var$; $combs_var$; $prevComb_q1$; $prevComb_q1$.
Devuelve: $combs_out$; $proxEstAut_out$; $esDep_out$; n_r.

1. **si** $(estAut_var == Q0_ST)||(estAut_var == Q3_ST)$ **entonces**
2.   **si** $combs_var < prevComb_q1$ **entonces**
3. $combs_out = combs_var$
4. $proxEstAut_out = Q1_ST$
5. $n_r = 1$
6.   **sino**
7. $combs_out = combs_var - prevComb_q1$
8. $n_r = 0$
9.   **si** $estAut_var == Q0_ST$ **entonces**
10. $proxEstAut_out = Q2_ST$
11.   **sino**
12. $proxEstAut_out = Q4_ST$
13.   **fin si**
14.   **fin si**
15. **sino si** $estAut_var == Q1_ST$ **entonces**
16. $combs_out = combs_var$
17. $proxEstAut_out = Q3_ST$
18. $n_r = 0$
19. **sino**
20. $combs_out = combs_var$
21. $proxEstAut_out = Q1_ST$
22. $n_r = 1$
23. **fin si**

rendimiento y corrección de la solución. Es por ello que, en caso de que ninguno de las cuatro números aleatorios en cola verifiquen la condición, se elegirá el menor de ellos, al que se le sumará $NBVariants$ y se descartará el bit de desbordamiento. En la implementación actual, se ha optado por primar el rendimiento, por lo que se ha seleccionado la segunda opción por defecto. No obstante, el sistema ofrece al usuario la posibilidad de seleccionar la otra opción si lo desea. Respecto a la opción por defecto, aunque el resultado sea válido, se tiene como inconveniente que la distribución que modeliza no es puramente equiprobable, ya que los números más pequeños poseen mayor probabilidad que el resto. Para el cómputo de $Variant$ es necesario, además del número aleatorio y los valores intermedios calculados anteriormente, conocer qué estado del autómata representa la unidad funcional. Para ello se hace uso de la señal $currentAutState$.

Aunque el Algoritmo 3.6 comienza con una división del conjunto de reglas en k secuencias independientes, es preciso notar que el sistema no efectúa una división explícita, sino que emplea las señales generadas por el anterior bloque, junto con la propagación de la señal dep_{rj} entre las unidades funcionales, calculando la división de modo implícito en la lógica de control implementada en el sub-módulo $ControlSubBlock_r$. De ese modo, se elimina un paso del algoritmo, salvando un ciclo de reloj.

Como resultado, el bloque de asignación genera el multiconjunto de aplicabilidad de reglas R siguiendo una distribución pseudo-multiprobable en dos ciclos de reloj, ofreciéndolo como entrada al siguiente bloque que compone la arquitectura Almond PS, el bloque de aplicación.

3.4.4. Bloque de aplicación

Una vez obtenido el multiconjunto de aplicabilidad, R, en el bloque de asignación, es preciso aplicar las reglas de acuerdo a las multiplicidades obtenidas. Este bloque recibe como entrada los índices de aplicabilidad de cada regla, y obteniendo como resultado un multiconjunto de objetos que representa las variaciones entre la configuración actual, C_i, y la resultante de aplicar R, C_{i+1} (Fig. 3.6). Nótese que el bloque que le sucede, el bloque de persistencia, recibe como entrada esta variación, y es en ese bloque donde realmente se obtiene la siguiente configuración,

C_{i+1}.

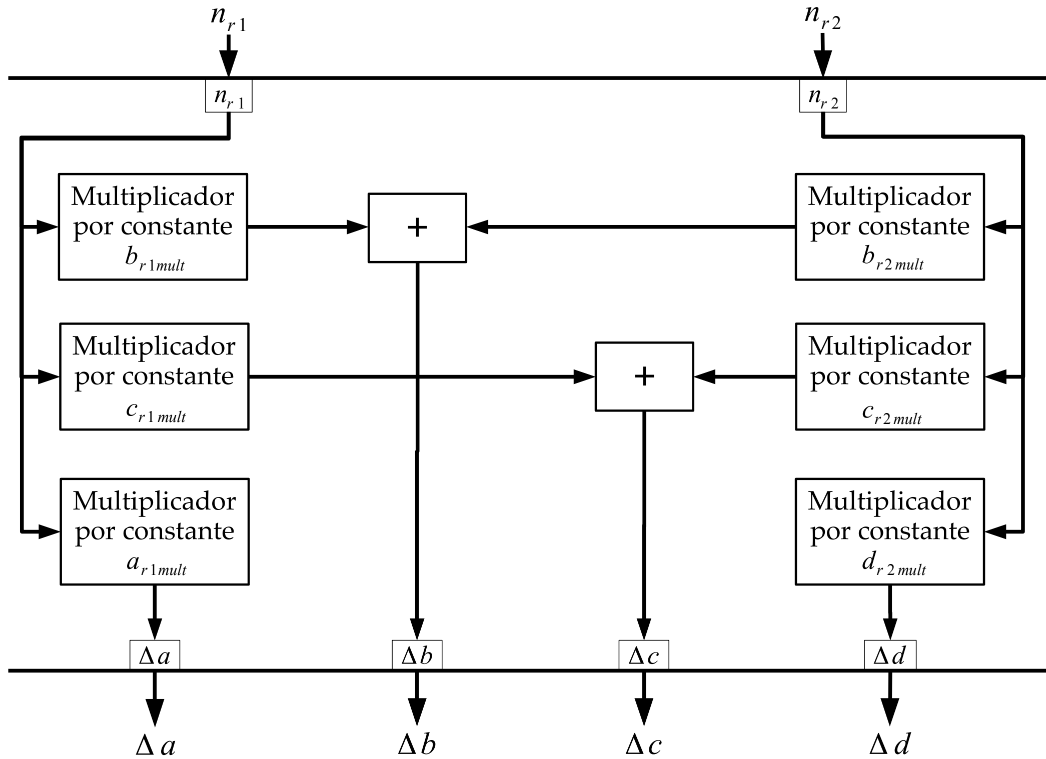


Fig. 3.6: Esquema del bloque de aplicación. Para facilitar la comprensión se ha preferido utilizar un ejemplo, en el que $R = \{r1, r2\}$, con $r1 : (a, b) \rightarrow (b, c)$ y $r2 : (b, c) \rightarrow (c, d)$.

Tal y como sucede con los bloques anteriores, el tamaño de los buses de datos y el número y características de sus componentes funcionales dependen del sistema P de entrada. Independientemente de los parámetros establecidos por este, considerando la definición de una regla, $r : u \rightarrow v$, este bloque posee dos etapas bien diferenciadas. Atendiendo a la primera, para cada regla es necesario obtener el número de objetos a los que afectará su aplicación, para lo que es preciso calcular el producto de las multiplicidades de cada uno de los objetos por el índice de aplicabilidad de la regla. En la segunda etapa, estos resultados deben ponerse en común con el resto de reglas del sistema empleando sumadores: los objetos situados en la parte izquierda de la regla, $o_i \in u$, son sustraendos en esta operación y los situados en la parte derecha, $o_i \in v$, sumandos. Para aquellos casos en los que el mismo

objeto aparezca en ambos extremos de la regla, durante la generación del sistema se calcula la variación total sobre la multiplicidad de ese objeto que tiene lugar al ejecutarse la regla.

Es preciso tener en cuenta que, aunque las multiplicidades de los objetos para una configuración son siempre mayor o igual que cero, la aplicación de una transición puede suponer la disminución de objetos en el sistema, por lo que los valores calculados, incluyendo la interfaz de salida del bloque, se encuentran en formato de números con signo en notación $Ca2$.

3.4.5. Bloque de entrada/salida

Respecto al bloque de entrada/salida, no se ha dotado al sistema de un bloque enfocado a una fase de producción, sino que se delega para un futuro su desarrollo, en función de las necesidades concretas del entorno. En este sentido, el enfoque modular de la arquitectura permite la encapsulación de este componente. Así, el desarrollo únicamente debe implementar la interfaz descrita a continuación, no siendo necesario conocer el funcionamiento del resto del sistema.

Dirección	**Tamaño**	**Bloque**	**Señal**
IN	32b	Control	Número de transición actual
IN	4b	Control	Estado de ejecución
IN	1b	Control	Fin de ejecución
OUT	S.I.	Control	Nueva semilla
OUT	4b	Control	Comandos de control
OUT	32b	Control	Nueva transición de parada
OUT	1b	Control	Ejecutar orden
OUT	1b	Control	Nuevo parámetro
IN/OUT	S.I.	Persistencia	Configuración actual

S.I.: Según Implementación.

Tabla 3.1: Interfaz del Bloque de Entrada/Salida.

El bloque de señales que definen la interfaz de este bloque se detalla en la Tabla 3.1. Su finalidad es la de ofrecer soporte para la lectura de resultados y control de las ejecuciones. Respecto al tamaño de las señales, en algunas ocasiones depende de las características del sistema de entrada, siendo responsabilidad del *software* de generación el configurar el número de bits adecuado.

Codificación	Significado
0000	Reiniciando
0001	En espera de iniciar computación
0010	En ejecución
0011	Ejecución pausada
0100	Fin de computación. Parada total
0101	Fin de computación. Parada adulta
0110	Fin de computación. Parada límite de transiciones
0111	Ejecución pausada
1000	Esperando comando
1001	Ocupado
1010 - 1111	Libres

Tabla 3.2: Códigos de estado del sistema ofrecidos por el Bloque de Control.

Del conjunto de señales que definen la interfaz del módulo, existen dos buses con un enfoque más general que el resto. El primero de ellos codifica el estado de la ejecución actual, según la Tabla 3.2. El segundo está dedicado al control del sistema, mediante comandos de control detallados en la Tabla 3.3.

Codificación	Significado
0000	Reiniciar sistema
0001	Iniciar ejecución
0010	Pausar ejecución
0011	Inicializar semilla
0100	Nueva semilla
0101	Establecer límite de transición
0110	Activar límite de transición
0111	Leer configuración actual
1000	Escribir configuración actual
1001 - 1111	Libres

Tabla 3.3: Códigos de control generados por el Bloque de Entrada/Salida.

Para la verificación de los resultados, se han empleado los *IPCore* de depuración de Xilinx. Para satisfacer las necesidades de este componente es preciso emplear 3 *IPCore* distintos: *Integrated Logic Analyzer* (ILA), *Virtual Input/Output* (VIO) y *ChipScope Integrated Controller* (ICO). En el caso del primero de ellos, el ILA, se trata de un bloque de monitorización de señales internas de la FPGA, usado

para recoger la configuración actual del sistema. El VIO permite leer y escribir registros en la FPGA, empleándose para el resto de las señales de control. Por último, el ICO es un *IPCore* encargado de la comunicación de los dos anteriores y el *software* de depuración XILINX CHIPSCOPE[2].

3.4.6. Bloque de control

Este bloque implementa la lógica de control del sistema empleando una máquina de estados. No supone ciclos adicionales durante la operación normal, ya que sus operaciones controlan el funcionamiento de los distintos bloques de la arquitectura Almond PS.

Aparte de los valores de la configuración actual, almacenados en el bloque de persistencia, es necesario almacenar el número de la transición en la que se encuentra actualmente el sistema. Esta funcionalidad es implementada en la lógica de control empleando un contador ascendente, que se incrementa tras finalizar cada transición, concretamente tras almacenar en el bloque de persistencia la configuración obtenida en la última transición.

Por último, tal y como se detalló en la Sección 2.1.1, existen distintas condiciones de parada. La arquitectura Almond PS ofrece soporte para:

- Ejecución de un número específico de transiciones. Por ejemplo, se considera que la computación finaliza al alcanzar la transición número 1000.

- El sistema converge hacia una configuración de parada total. Esto es, se alcanza una configuración para la que el conjunto *Appl* es nulo.

- El sistema converge hacia una configuración de parada total adulta. Esto es, a pesar de que el conjunto *Appl* no sea nulo, la configuración actual no cambia independientemente del multiconjunto de aplicabilidad seleccionado. Con el objetivo de reducir la complejidad del sistema y los recursos utilizados, se ha considerado una aproximación a la condición de parada teórica. En ella se considera que el sistema converge hacia una configuración de parada adulta, cuando la aplicación de las últimas j transiciones consecutivas, ha generado la misma configuración. Este parámetro es seleccionable por el usuario.

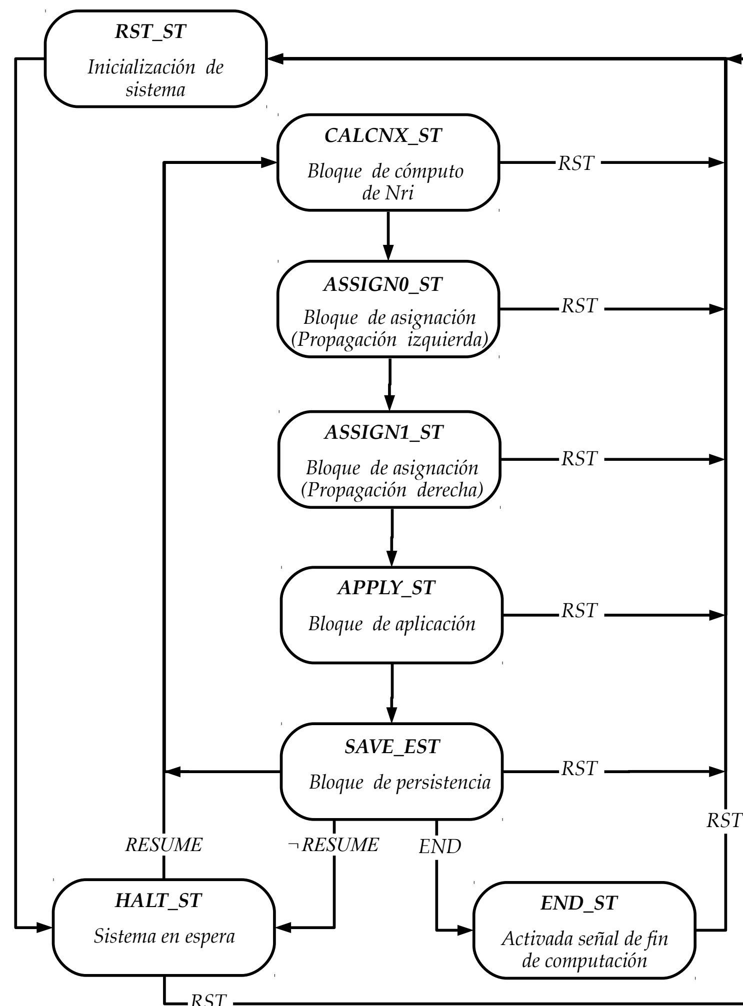


Fig. 3.7: Diagrama de estados simplificado de la unidad de control. Junto al nombre del estado aparece una descripción con el bloque activo en ese momento o la tarea que se realiza. En las transiciones únicamente aparecen aquellas señales de relevancia.

Respecto a los detalles de implementación, los estados de control se agrupan en: estados asociados a la ejecución ordinaria del sistema, estados asociados a la comunicación con el *software* de control y estados de inicialización y parada (Fig. 3.7). Respecto al último grupo, se distinguen los estados RST_ST, $HALT_ST$ y END_ST. En RST_ST se produce la inicialización del sistema, consistente en restaurar la configuración inicial (bloque de persistencia) e inicializar el generador de números aleatorios (bloque de asignación) y los registros de control asociados a la ejecución, entre ellos el que almacena el número de transiciones. En $HALT_ST$ el sistema permanece en pausa, manteniendo el estado actual. Por último, el estado END_ST es similar al anterior, con la particularidad de que activa la señal de fin de ejecución.

Por último, existen cinco estados asociados a la ejecución de un sistema P, relacionados directamente con la división empleada en la arquitectura Almond PS, así como con el funcionamiento de cada bloque, generando las señales de control requeridas. Así, *CALCNX_ST* genera las señales de control del bloque de cómputo de N_{r_i}; *RULECALC0_ST* y *RULECALC1_ST* las del bloque de asignación, concretamente, *RULECALC0_ST* activa la propagación izquierda, mientras que *RULECALC1_ST* hace lo propio con la propagación derecha; *APPLYRULE_ST* las del bloque de aplicación; y *UPDREG_ST* las del bloque de persistencia.

Debido a que el tiempo de ejecución de cada transición es constante, independientemente de los parámetros de entrada, en una ejecución ordinaria no se requiere ninguna señal de control como condición para cambiar de estado entre aquellos asociados a la ejecución del sistema. Estando en cualquiera de ellos, una señal de *reset* provoca una transición al estado *RST_ST* para posteriormente iniciar la ejecución. Por otro lado, el *software* puede pausar la ejecución, pasando al estado *HALT_ST* al finalizar la actual transición. Cuando algunas de las condiciones de parada son alcanzadas, el sistema pasa al estado *END_ST*, notificando el fin de la ejecución.

En relación a la funcionalidad asociada a la entrada/salida, es implementada con una máquina de estado adicional que ejecuta los comandos que recibe del Bloque de Entrada/Salida, al tiempo que ofrece información de estado. Los parámetros que son accesibles son la configuración actual, el número de transiciones y

[2] XILINX CHIPSCOPE: `http://www.xilinx.com/tools/cspro.htm`

el estado del sistema. Todos los parámetros son proporcionados por este bloque, a excepción de la configuración actual, que es solicitada por el Bloque de Entrada/Salida directamente al Bloque de Persistencia. Para la consulta del estado del sistema, existe un bit dedicado a informar del fin de la ejecución, así como un bus que indica el estado del sistema, empleando la codificación que se muestra en la Tabla 3.2. Respecto a la ejecución de comandos, un segundo bus dedicado ofrece un canal de comunicación que permite controlar la ejecución, así como configurar la semilla empleada en el generador de números aleatorios o la configuración inicial, según recoge la Tabla 3.3.

3.5. Implementando nuevos sistemas P

En una de las secciones anteriores se detallaron los sistemas P aceptados por la arquitectura *hardware*. Así, se decidió postergar el análisis de cómo podría afectar al diseño propuesto la implementación de un nuevo sistema P, en aras de mejorar la comprensión del documento, presentando, en primer lugar, una implementación concreta. Una vez descrita la arquitectura, en esta sección se detallan las distintas modificaciones previstas de la arquitectura Almond PS.

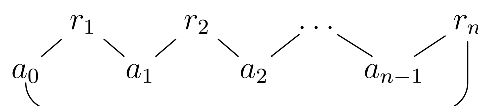
Tomando como punto de partida el entorno formal de redes de células citado y descrito con anterioridad, un nuevo sistema P podría diferenciarse en el modo de derivación, el modo de selección de un elemento del conjunto *Appl*, el grafo de dependencia de las reglas del sistema y la condición de parada. En este sentido, es preciso destacar que nuevos tipos de reglas y de objetos no suponen un impacto en la arquitectura Almond PS, al ser abstraídos por el entorno formal de redes de células. Generalizando, ninguno de los puntos de variación mencionados suponen impacto alguno en el modo de persistir las distintas configuraciones del sistema, de calcular el número máximo de aplicabilidad de cada regla y la forma en la que se aplica una transición, por lo que el bloque de persistencia, el bloque de cómputo de N_{r_i}, el bloque de aplicación y el bloque de entrada/salida no son candidatos a ser modificados al implementar un nuevo tipo de sistema P.

La modificación del modo de derivación únicamente afecta al bloque de asignación. Ya que el conjunto *Appl* variará, y por lo tanto, el lenguaje que lo representa será distinto. Más adelante se aportará más información en este sentido. Tal y

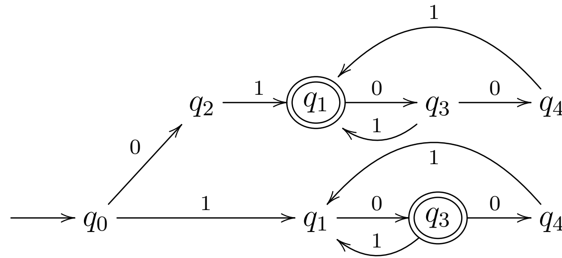
como se mencionó al describir el bloque de cómputo de N_{r_i}, este módulo puede ser modificado según el modo de derivación, únicamente con el propósito de optimizar el consumo de recursos y la ejecución del sistema. Así, atendiendo a la implementación presentada a lo largo de este capítulo, se presenta una solución genérica, válida para cualquier modo de derivación, y una concreta, específica del modo de derivación implementado, *smax*.

Atendiendo a la introducción de un nuevo modo en el que es seleccionado, del conjunto *Appl*, el multiconjunto que será aplicado en cada transición, la modificación necesaria en el sistema se localiza en el submódulo GeneradorPRN del bloque de asignación. En este sentido será necesario modificar la distribución que sigue el generador de números aleatorios, y/o introducir elementos funcionales, en caso de que el elemento seleccionado dependa de la configuración del sistema, como por ejemplo, del número de configuración en la que se encuentre.

El siguiente punto de variación lo constituye el grafo de dependencias de las reglas del sistema. En este sentido, los cambios también afectarían al bloque de asignación, tal y como sucede al variar el modo de derivación del sistema: modificar el lenguaje que representa el conjunto *Appl*, supone calcular un nuevo autómata, y una nueva serie formal de potencias asociada. A modo de ejemplo, se considerará el conjunto de reglas que forman un grafo de dependencia circular para un sistema trabajando en modo *smax*.

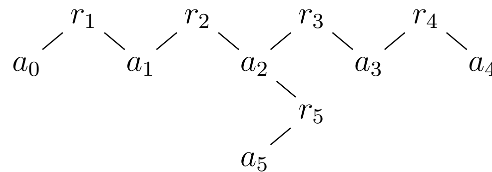


Únicamente nos centraremos en el caso que presenta interés, aquel en el que todas las reglas son aplicables una sola vez, tal y como se describe en la Sección 3.3, concretamente la tercera casuística del modo *smax*, aquella en la que $N_{a_i} = 1$. En este caso, el vector del conjunto de aplicabilidad de estas reglas (es decir, cadenas binarias de longitud n con valor 1 en la posición i-ésima correspondiente a la elección de la regla r_i) puede describirse empleando las palabras de longitud n del siguiente autómata:



Este autómata se obtiene para el lenguaje L_I, partiendo del ejemplo 2 y añadiendo una condición adicional: si se elige la regla r_1 entonces no se elige r_n, y viceversa.

Empleando los procedimientos e ideas presentados a lo largo del capítulo, es posible describir con lenguajes regulares secuencias de reglas que formen estructuras de mayor complejidad. Por ejemplo, la siguiente estructura



puede representarse mediante un lenguaje regular sobre el alfabeto binario si el número de símbolos a_2 es conocido. Dicho lenguaje puede construirse de manera similar al lenguaje anterior, que consideraba la dependencia circular. Así, es posible calcular la función $NBVariants(\Pi, C, smax)$, seleccionando, en primer lugar, al autómata adecuado, según el valor de N_{a_2}, para posteriormente calcular su función de generación. De ese modo, este procedimiento se puede extender para describir lenguajes regulares para la aplicabilidad de reglas que presenten un grafo de dependencia sin ciclos en las intersecciones. Al igual que para los ejemplos anteriores, estos procedimientos formales pueden ser implementados en tecnología FPGA, presentando un tiempo de ejecución constante.

Por último, la modificación de la condición de parada únicamente puede afectar al bloque de control. En este sentido, la arquitectura desarrollada ofrece soporte para distintas condiciones. No obstante, si se desea introducir un nuevo criterio, los cambios se reducen a implementar las condiciones lógicas asociadas en el bloque de control.

3.6. Conclusiones

En este capítulo se ha presentado la arquitectura Almond PS, una implementación de los sistemas P estáticos, cuyo conjunto de aplicabilidad, $Appl(\Pi, C, \delta)$, corresponde a un lenguaje no-ambiguo libre del contexto. Se trata, en consecuencia, de una implementación con un enfoque no contemplado en las implementaciones *hardware* existentes, las cuales se encontraban muy acopladas al modelo teórico de la computación con membranas. Este nuevo punto de vista es fruto de la búsqueda de un marco algorítmico que permitiera desarrollar implementaciones que pudieran obtener el máximo rendimiento posible de la tecnología FPGA. Como consecuencia directa de este planteamiento, la arquitectura Almond PS es capaz de completar una transición en tan solo 5 ciclos de reloj, independientemente del tamaño del sistema o de sus elementos estructurales, esto es, tipos de objetos o reglas, ofreciendo una potencia computacional muy próxima al valor ideal, de un ciclo por transición.

El segundo aspecto destacable es el hecho de que, a diferencia del resto de implementaciones actuales, el rango de sistemas P que puede simularse sólo se ve afectado por la dependencia entre las reglas. Se trata de una consecuencia directa del enfoque empleado, en el que cada uno de los elementos del conjunto de aplicabilidad es representado como palabras de un lenguaje regular y libre de contexto. Partiendo de esta codificación, es posible calcular el tamaño del conjunto de aplicabilidad empleando series formales de potencias. A continuación, basta con seleccionar aleatoriamente un número i, comprendido entre 0 y ese tamaño para, finalmente, generar el elemento i-ésimo.

Además, durante el diseño de la arquitectura se ha considerado que los sistemas P constituyen un modelo de computación orientado a máquinas, requiriendo cada instancia concreta una máquina distinta. Es por ello que Almond PS es una arquitectura donde se ha primado la flexibilidad durante la fase de diseño. En este sentido, se ha desarrollado un sistema altamente modular, con interfaces claramente definidas, presentando cada uno de los bloques una encapsulación que independiza su funcionamiento del resto de los componentes. Además, toda la lógica dependiente del sistema P se ha encapsulado en el Bloque de asignación, reduciendo al mínimo el impacto de añadir soporte para una nueva relación de

reglas, aumentando el número de sistemas soportados.

Por último, la implementación llevada a cabo genera sistemas P con un modo de derivación *smax*, y cuyo grafo de dependencias de las reglas tiene forma de cadena. Así, se ha presentado la metodología a seguir para implementar un modo de derivación y/o un nuevo grafo de dependencias entre reglas: (1) modificar el lenguaje que representa el nuevo conjunto *Appl*; (2) diseñar un nuevo autómata; (3) generar la serie formal de potencias asociada; y (4) diseñar el nuevo bloque de asignación y sustituir el existente. Por tanto, no es necesario volver a diseñar una nueva implementación, sino únicamente sustituir un módulo de esta, con sus interfaces claramente definidas. A diferencia de otros desarrollos, una vez añadido un modo de derivación y un grafo de dependencias entre reglas, el resto de puntos de variación de un sistema P no afecta a la implementación, que es capaz de ejecutar cualquier sistema P estático con el mismo grafo de relación entre reglas y modo de derivación.

Capítulo 4

Software de generación de sistemas P

La generación automatizada de instancias concretas es un elemento necesario en la implementación de sistemas P. Así, es necesario el desarrollo de *software* que ofrezca esta funcionalidad. Al mismo tiempo, son las mismas particularidades de estos sistemas, las que requieren que los principios de diseño de flexibilidad y extensibilidad deban trasladarse a estas herramientas.

Este capítulo describe el avance en este ámbito, al tiempo que ofrece herramientas para automatizar la aplicación de la metodología de desarrollo basado en pruebas, muy asentada en el mundo del desarrollo *software*. De ese modo, una primera parte describe la implementación inicial, separando la entrada de sistemas P y el análisis y generación *hardware*. A continuación, describe tareas enfocados a la evolución y mejora del trabajo presentado aplicando la tecnología *Model-Driven Engineering* (MDE), ofreciendo una metodología para la aplicación, parcialmente automatizada del desarrollo basado en pruebas. Por último, el capítulo finaliza con las conclusiones más importantes del trabajo realizado.

4.1. Procesamiento de los sistemas de entrada

El primer paso de todo el procesamiento lo constituye la entrada del sistema P, así como su posterior análisis y representación.

La aplicación recibe el sistema P como un fichero de entrada, utilizando el lenguaje desarrollado para *P-Lingua*, habilitando algunas extensiones para soportar los conjuntos P y Q del entorno de red de membranas. Para su implementación se ha empleado el entorno *Xtext*[1], que permite, a partir de un modelo de especificación, el desarrollo de lenguajes específicos de dominio, generando el código necesario para la representación del *Abstract Syntax Tree* (AST), que captura la estructura sintáctica de la entrada, así como *plugins* para personalizar el *Integrated Development Environment* (IDE) ECLIPSE[2]. Este punto de partida nos permite hacer uso de las herramientas desarrolladas por otros grupos de la disciplina, así como la integración de soluciones en un futuro.

A continuación, el AST es transformado a una representación propia, que constituye el punto de entrada para los componentes de generación *hardware*. Durante esta transformación y el proceso anterior, se comprueba que no existen errores sintácticos o semánticos en el fichero de entrada.

La implementación inicial consiste en un desarrollo *software* a medida, que permite validar las hipótesis de diseño desde un punto de vista funcional. Además, ha sido empleado durante el desarrollo para la ejecución de las pruebas unitarias y de integración de la arquitectura Almond PS.

En la representación empleada, mostrada en la Fig. 4.1, un sistema P está representado por la clase InstancePS. Esta clase contiene métodos de consulta básicos que permiten acceder al tipo de objetos del sistema y a sus reglas, así como de consultas relacionadas con las dependencias entre reglas y objetos. Dado un sistema P, este queda determinado por una configuración inicial, una estructura y un modo de derivación. A su vez, la estructura describe los objetos presentes en el sistema y las reglas que contribuyen a su definición funcional. En este sentido, los objetos están modelados por la clase *ObjType*, en base a un identificador numérico y una representación legible. Esta clase es extendida con *AdvObjType*, con el propósito de mejorar los tiempos de ejecución al hacer un seguimiento de las reglas que lo consumen y producen. Este enfoque se mantiene en la clase *Rule*, representación de una regla del sistema. Posee un identificador numérico y un nombre, además de almacenar información sobre los objetos que consume y produce,

[1] *Xtext*: `https://eclipse.org/Xtext/`
[2] https://eclipse.org/

y proveer de servicios de consulta de estos parámetros. De ese modo, la estructura es representada por la clase *Structure*, que posee objetos de tipo *AdvObjType* y *Rule*. Atendiendo a la configuración del sistema, esta es representada con la clase *Configuration*, con la finalidad principal de almacenar los objetos existentes en un momento concreto de la ejecución.

La transición constituye el último elemento de los enumerados anteriormente. Es el componente encargado de llevar a cabo, en última instancia, el aspecto funcional del sistema, tomando como datos de entrada los elementos anteriores: la configuración y su estructura, esto es, los objetos presentes y las reglas. *ITransition* define la interfaz de los servicios que deben proporcionar las clases que modelen este elemento. En este sentido, la transición *smax* está modelada por la clase *TransitionMin1_Chain*. Tomando como objetivo la fase de pruebas de la arquitectura *hardware*, la implementación de la interfaz se organiza siguiendo la división estructural de Almond PS. Esto permite emplear el mismo código para las pruebas funcionales iniciales, y las posteriores pruebas unitarias y de integración de toda la arquitectura.

En resumen, un sistema P es representado por la clase *InstancePS*. Los tipos de objetos y reglas que lo definen son modelados por objetos *AdvObjType* y *Rule*, respectivamente, asociados a un objeto *Structure* de *InstancePS*. Atendiendo al aspecto funcional, la clase *Configuration* representa una configuración del sistema, y cada tipo de derivación que se desee implementar debe satisfacer la interfaz *ITransition*. Debido a que el sistema está enfocado hacia el diseño y validación del desarrollo *hardware*, el punto de entrada funcional del componente es la interfaz *ITransition*, con soporte para su validación funcional. Otras clases de utilidades completan el desarrollo *software*, como son aquellas destinadas a la generación aleatoria de casos de pruebas o persistencia de estos.

4.2. Componentes de generación *hardware*

Una vez obtenida una representación de la instancia del sistema P a implementar, es necesario analizar, parametrizar y, en último lugar, generar el código a sintetizar en la FPGA. A continuación se detalla como se estructura el código que ofrece esta funcionalidad para, posteriormente, describir el desarrollo específico

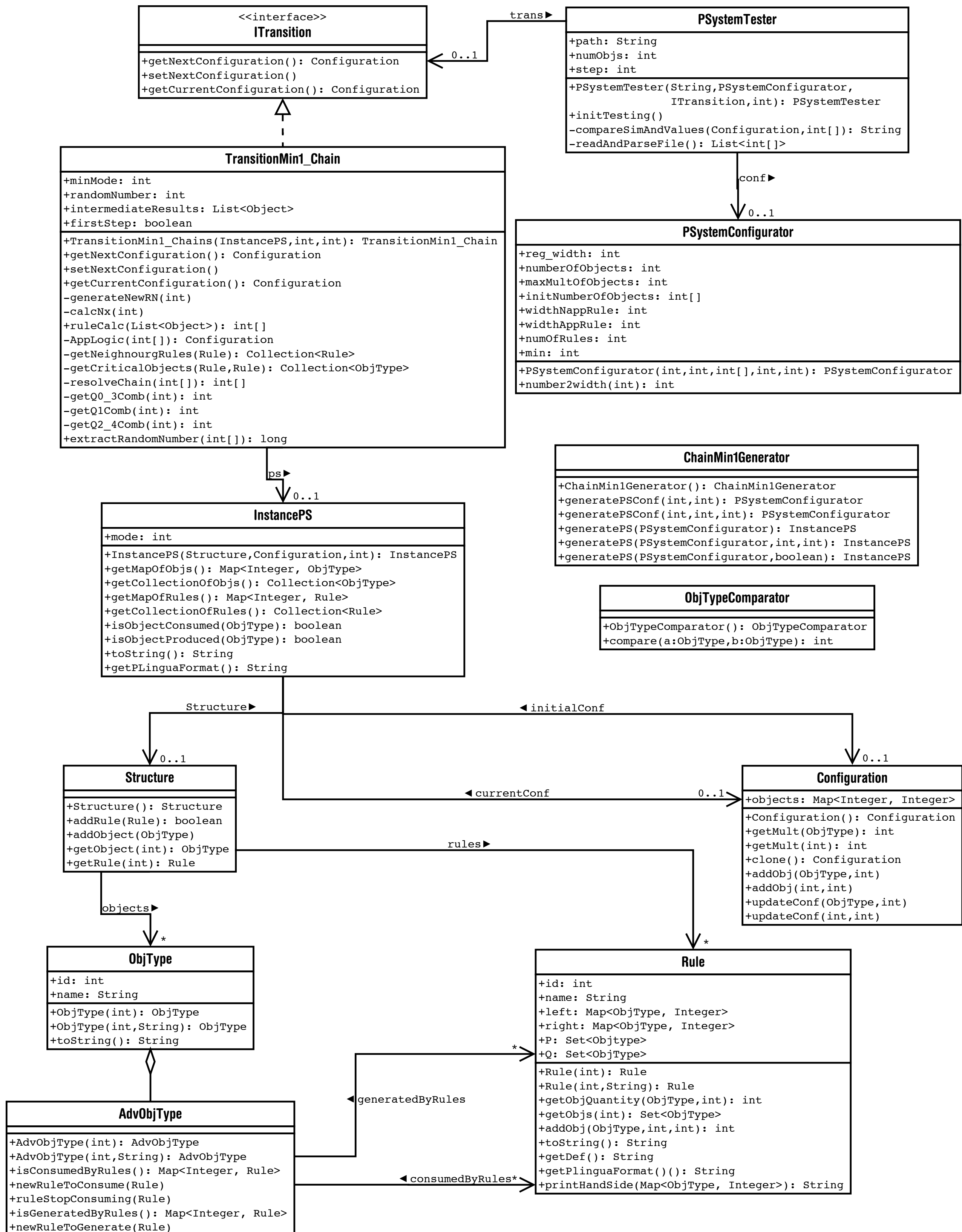


Fig. 4.1: Diagrama UML simplificado de la solución *software* de generación.

para Almond PS.

4.2.1. Estructuración de los módulos específicos de generación

El componente *software* generador (Fig. 4.2), se organiza en torno a la clase *CodeGenerator*. Esta clase actúa como un controlador central, en el que se registran los distintos componentes que describen las instrucciones de generación para cada uno de los módulos *hardware*. Además, es la encargada de dar soporte a aspectos generales del procesado de código, como la generación de la estructura de ficheros, ficheros auxiliares del proyecto y aquellos específicos de las herramientas de generación.

La funcionalidad necesaria para la generación de cada componente *hardware* está dividida en dos interfaces: *IModuleGeneratorControler* e *IModuleGenerator*. La primera de ellas define los servicios auxiliares necesarios para la generación del código del mismo, como ficheros auxiliares, opciones de las herramientas de síntesis exclusivas del componente o configuración de la propia generación de código. Respecto a *IModuleGenerator*, es el encargado de la generación estricta del código HDL. En este caso, se ha optado por VHDL, generando al menos una sección *Package* y una *Entity* por cada elemento.

En consecuencia, para añadir un nuevo componente, o sustituir uno existente, es necesario implementar el conjunto de los servicios ofrecidos por las dos interfaces anteriores. En este sentido, se han implementado dos clases abstractas, *ModuleGeneratorControler* y *ModuleGenerator*, que implementan estas interfaces, cubriendo un comportamiento estándar base. Respecto a la primera, ofrece métodos para la generación de ficheros, directorios, raíz de generación, acciones posteriores al procesado y opciones de síntesis y otros parámetros específicos de las herramientas de generación *hardware*. Atendiendo a la segunda, se encuentra acoplada al lenguaje VHDL, ofreciendo una estructura para la generación de los distintos elementos requeridos por el lenguaje. Así, para cada módulo se genera un bloque *Entity* y una *Architecture*, ofreciendo la opción de generar uno de tipo *Package*.

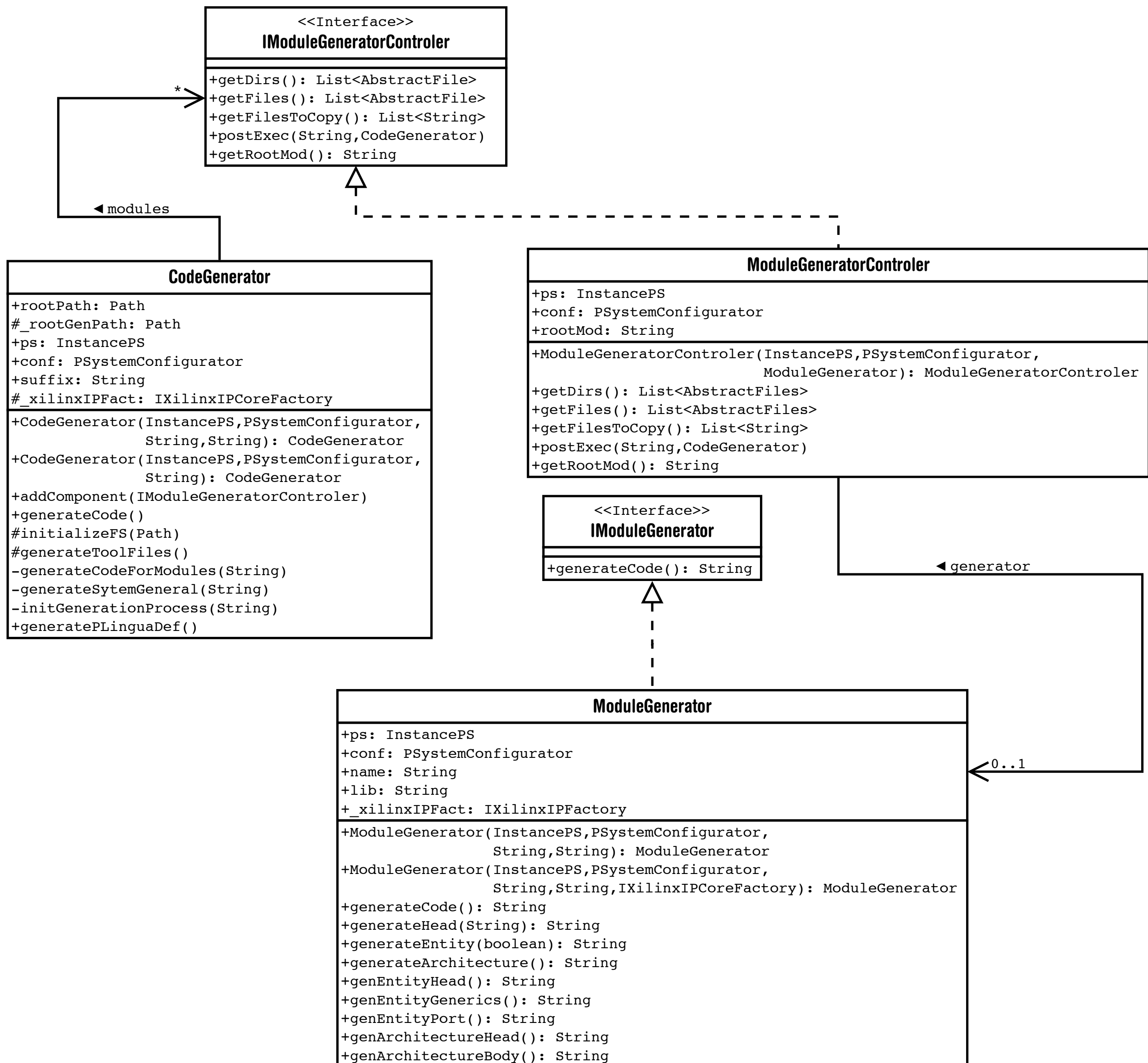


Fig. 4.2: Diagrama UML simplificado de la estructura base de la herramienta de generación de código.

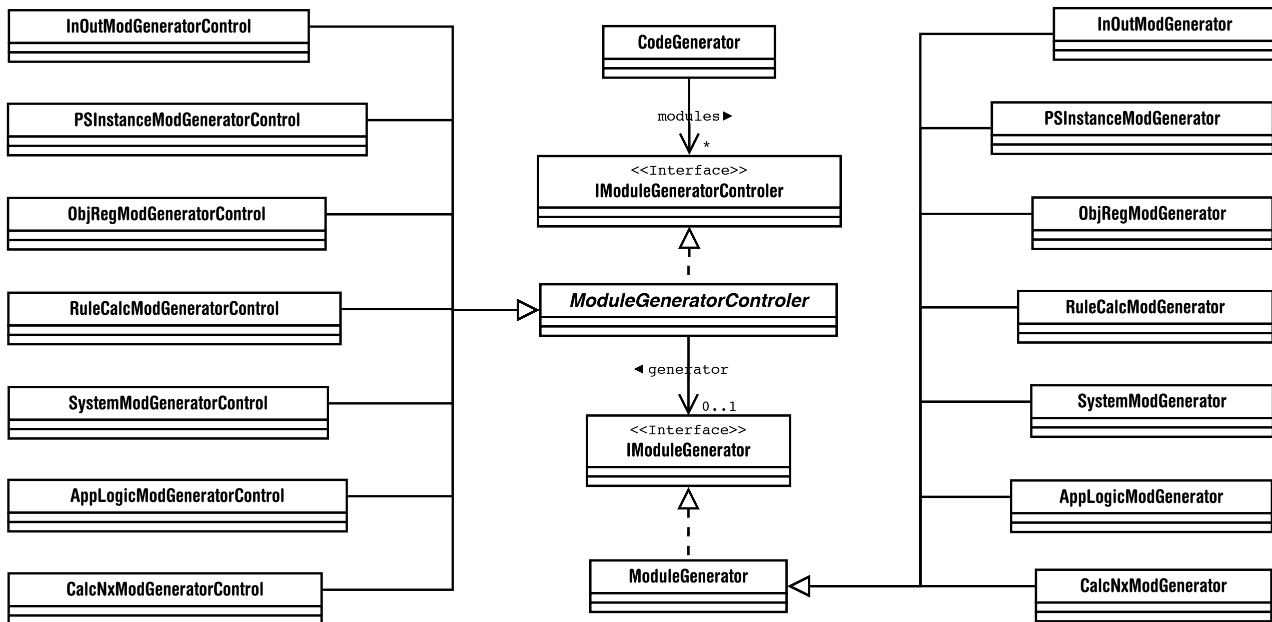


Fig. 4.3: Diagrama UML simplificado de la solución *software* de generación de código de la plataforma Almond PS.

4.2.2. Elementos generadores de Almond PS

Tomando como base la estructura de clases anterior, se describe a continuación el código necesario para la implementación de la arquitectura Almond PS. Así, está formada por seis bloques independientes (Fig. 3.2): bloque de entrada/salida, bloque de persistencia, bloque de cómputo de N_{r_i}, bloque de asignación, bloque de aplicación y bloque de control. En la Fig. 4.3 se muestran las clases que generan la implementación de cada uno de los bloques que forman la arquitectura Almond PS. A continuación, pasamos a describirlos.

Las clases *ObjRegModGeneratorControl* y *ObjRegModGenerator* ofrecen la funcionalidad necesaria para el procesamiento del bloque de persistencia, integrando la definición de otros componentes básicos, como registros o sumadores, que son añadidos al proyecto.

Respecto al bloque de cómputo de N_{r_i}, su generación es responsabilidad de las clases *CalcNxModGeneratorControl* y *CalcNxModGenerator*. Como en el componente anterior, es necesaria la inclusión de otros elementos base, destacando los

divisores. En este sentido, es necesario hacer una división por cada elemento de la parte izquierda de cada regla. Las reglas son un componente estructural estático, por lo que el divisor es contante durante toda la ejecución. Por lo tanto, no se precisa de un divisor completo, pudiéndose simplificar el problema a la implementación de la operación de división con un divisor constante. El uso de multiplicadores, en el que uno de los factores es el inverso del divisor, es una implementación adecuada a este problema, con la ventaja de requerir menos recursos *hardware* o temporales. De ese modo, la división es realizada a través de multiplicadores, empleando los *IP Core* propietarios de XILINX, el *IP Core xilinx.com:ip:mult_gen:11.2* en el caso de las FPGA VIRTEX7. Estos *IP Core* son generados por la herramienta XILINX CORE GENERATOR SYSTEM[3], del mismo fabricante, a partir de un fichero .xco con los valores parametrizados del bloque a generar, y uno .cgp con el entorno objetivo de implementación, como son el modelo de FPGA y varias opciones de generación adicionales. Ambos ficheros son producidos por la funcionalidad contenida en la clase *ConstDividerModGenerator*, con los parámetros calculados por *ConstDividerModGeneratorControl*, que adicionalmente debe obtener el inverso del divisor con la precisión adecuada, configurando un clúster de bloques multiplicadores, en función de los parámetros del sistema P, la FPGA objetivo y algunos valores adicionales. Estas clases no se muestran en la Fig. 4.3, al tratarse de un submódulo de los módulos principales de la arquitectura.

El bloque de asignación es el más complejo de la arquitectura, además, es donde se encapsula la funcionalidad dependiente de las relaciones existentes entre las distintas reglas. Es por ello que las clases que contemplan su funcionalidad, *RuleCalcModGeneratorControl* y *RuleCalcModGenerator*, deben obtener la lógica del autómata para, a continuación, generar el código HDL asociado a los componentes requeridos. Toda esta funcionalidad es ofrecida en la clase *RuleCalcModGeneratorControl*, que genera la lógica necesaria para las dependencias entre reglas detalladas en este documento. El resultado es una *Arithmetic Logic Unit* (ALU), siendo sus principales componentes sumadores y multiplicadores. Además de los bloques mencionados, es esta unidad la que recoge el no determinismo de la solución, por lo que es necesario la generación de números aleatorios, por lo que se ha contemplado la inclusión de un generador de números

[3]XILINX CORE GENERATOR SYSTEM: `http://www.xilinx.com/tools/coregen.htm`

aleatorios LFSR. Este componente, al igual que el resto, puede ser modificado o sustituido por otro, modificando el parámetro de configuración correspondiente en *RuleCalcModGeneratorControl*.

El último paso del *pipeline*, el bloque de aplicación, es obtenido gracias a las clases *AppLogicModGeneratorControl* y *AppLogicModGenerator*. Al igual que el anterior, se trata de una ALU, aunque de menor complejidad, constituida por sumadores y multiplicadores.

Respecto al bloque de entrada/salida, en la implementación actual se ha optado por emplear los bloques de comunicación de XILINX: ILA, ICO y VIO. Así, *InOutModGeneratorControl* genera los parámetros dependientes de la instancia del sistema P, considerando la interfaz definida con el resto del sistema. Es *InOutModGenerator* el responsable de la generación, en base a los parámetros recibidos de *InOutModGeneratorControl*, de los ficheros parametrizados que necesitan las herramientas del fabricante para la síntesis de estos *IP Core*, así como de la lógica de pegamento necesaria.

En el *software* de generación, la funcionalidad de interconexión de los distintos elementos de la plataforma *hardware* está encapsulada junto con la lógica de control de esta. Las clases responsables son *PSInstanceModGeneratorControl* y *PSInstanceModGenerator*. Entre los principales componentes se encuentran los registros que definen las etapas, y la máquina de estados que gobierna el funcionamiento de todo el sistema. Por último, es necesaria la generación de elementos auxiliares, como el módulo que se ocupa de la señal del sistema, o constantes de generación empleadas en el código HDL. Esta funcionalidad se encuentra implementada en las clases *SystemModGeneratorControl* y *SystemModGenerator*, que se encargan, adicionalmente, de encapsular todos los elementos *hardware* producidos por las anteriores clases en el componente *hardware* final, al que se conectan las señales de entrada/salida de la FPGA y el módulo de reloj del sistema.

4.3. Mejora del *software* de generación con MDE

Tal y como se ha comentado anteriormente, es responsabilidad de dos clases la generación de cada componente de la arquitectura, el controlador y el generador, generalmente denominados *XModGeneratorControl* y *XModGenerator*, siendo

X el nombre del componente. Sobre la primera de ellas recae la lógica de análisis y parametrización, y sobre la segunda la generación del código HDL. La primera iteración, contiene el código JAVA necesario para, siguiendo las instrucciones del controlador, construir el código desde cero, repartiendo la lógica entre distintas funciones, empleando gran número de instrucciones de manejo de cadenas de texto.

Aunque válido, este enfoque presenta un gran acoplamiento con los distintos módulos, siendo necesaria su reescritura casi por completo para cada uno de los bloques. Para eliminar estos problemas, se ha optado por el empleo de la tecnología MDE.

4.3.1. Breves nociones de MDE

La tecnología MDE [Schmidt, 2006] está enfocada en la creación, análisis y transformación de modelos de dominios, esto es, una representación abstracta de un dominio de aplicación particular. *Model-Driven Architecture* (MDA) [Miller and Mukerji, 2003] es una iniciativa del grupo *Object Management Group* (OMG)[4], y define una arquitectura que ha sido empleada como soporte para el *framework* de desarrollo *software*. En este documento se describen, brevemente, aquellos elementos esenciales para la comprensión del trabajo realizado. Se sugiere la siguiente bibliografía si se desea obtener una visión más completa de este paradigma: [Schmidt, 2006], [Miller and Mukerji, 2003] y [Steinberg et al., 2008] para su implementación en ECLIPSE.

MDA define un modelo como una representación de un sistema, entendiéndose este como cualquier cosa, como por ejemplo un entorno *software*, una arquitectura, un sistema P o un *System On Chip* (SOC). Contempla tres modelos, el *Computation Independent Model* (CIM), independiente de la computación, recoge las funcionalidades del sistema, pero no su modo de implementación; el *Platform Independent Model* (PIM), describe las operaciones del sistema pero independientemente de la plataforma; y el *Platform Specific Model* (PSM), que añade al anterior los detalles de una arquitectura concreta. De estos tres, nos centraremos en los dos últimos, el PIM y el PSM.

Una vez definidos los modelos, MDA establece transformaciones entre modelos,

[4]Web del grupo: http://www.omg.org/.

esto es, un procedimiento a través del cual es posible pasar de un modelo PIM a su correspondiente PSM. Esta transformación puede ser bidireccional, e incluso extender el patrón y contemplar varios modelos, con distintos niveles de abstracción, y con sus respectivas transformaciones, en vez de dos únicos niveles de abstracción. Así, dados dos modelos de consecutivos niveles de abstracción, el de mayor abstracción se denomina meta-modelo del de menor abstracción.

La Fig. 4.4 muestra un ejemplo de cómo podría aplicarse esta tecnología al área de la implementación de sistemas P. En este sentido, el mayor nivel de abstracción estaría representado por AbstractPS, abstracción de los distintos tipos de sistemas P. A partir de él, se pueden definir modelos intermedios que recojan particularidades de distintas plataformas de implementación, como *software*, GPU o *hardware*. A su vez, cada una de ellas cuenta con implementaciones concretas, por ejemplo Almond PS, definiendo un nivel de abstracción en base al lenguaje HDL empleado y, por último, en base al dispositivo FPGA. Así, la tecnología MDE aporta un *framework* de gran ayuda a la hora de unificar las herramientas e implementaciones existentes, no solo con otros trabajos que empleen tecnología FPGA, sino aquellos que empleen una tecnología diferente, como las basadas en procesadores de propósito general o GPU. Además, su aplicación aporta ventajas a la metodología de desarrollo *hardware*, al permitir la reutilización de código y la generación automática de patrones de pruebas.

La metodología de trabajo consiste en la definición de nuevos modelos, y de las transformaciones necesarias respecto a aquellos de nivel de abstracción adyacentes, aunque en la práctica es posible dar varios saltos con una única transformación. Con ello se obtiene un *framework* de análisis y generación de implementaciones de sistemas P con una gran portabilidad, interoperabilidad y reusabilidad.

OMG contempla tres estándares para la metodología con MDA: *Unified Modeling Language* (UML) [Siegel, 2015] para la especificación de modelos, *OMG's MetaObject Facility* (MOF) [Group, 2004] para la manipulación de modelos y *XML Metadata Interchange* (XMI) [Group, 2015] para el intercambio de información empleando *eXtensible Markup Language* (XML) [w3c, 2006].

Respecto a las tecnología empleadas, el ecosistema de Eclipse representa una de las iniciativas más activas. *Eclipse Modeling Framework* (EMF) [Steinberg et al., 2008], un *framework* de modelado y servicios de generación de código, que imple-

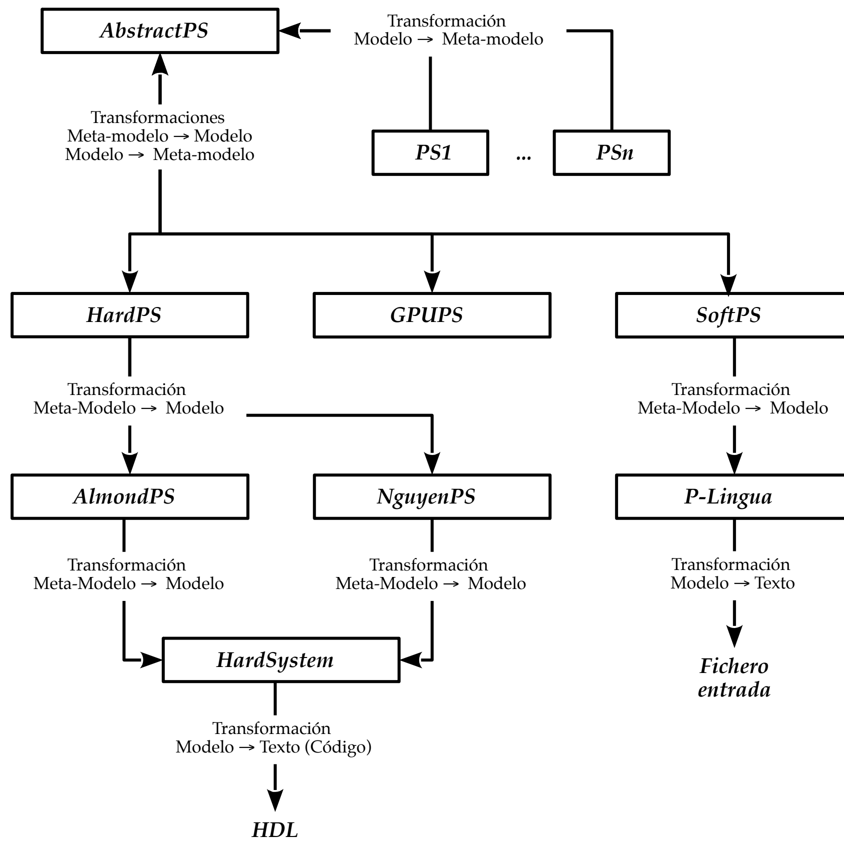


Fig. 4.4: Modelos y transformaciones entre modelos de un *framework* de simulación de sistemas basado en MDE. Las flechas representan transformaciones entre modelos.

menta parcialmente el estándar MOF. Haciendo uso de un meta-modelo, denominado Ecore, ofrece un elenco de herramientas *software* que permiten la generación de clases Java como implementación del modelo, además de un conjunto de clases que permiten la edición y visión de modelos, y otros servicios, como son notificaciones de modificación de modelos, persistencia o serialización XML, entre otros.

Por último, Epsilon [Kolovos et al., 2015] extiende algunas de las funcionalidades de EMF, proporcionando un conjunto de herramientas y lenguajes orientados a la manipulación de modelos. Partiendo de un lenguaje general, *Epsilon Object Language* (EOL), ofrece soporte, entre otros, para la transformación de modelos,

empleando el lenguaje *Epsilon Transformation Language* (ETL), su validación, con el *Epsilon Validation Language* (EVL) y la generación de código, a través del *Epsilon Generation Language* (EGL). De los anteriores, únicamente haremos uso de EGL, un lenguaje de transformación de modelo a texto, código en este caso, basado en plantillas. Como el resto de lenguajes de Epsilon, comparte el lenguaje EOL, más algunas funcionalidades específicas.

4.3.2. Generación de componentes *hardware* con MDE

Tomando como punto de partida el desarrollo *software* comentado anteriormente, el propósito es aplicar la tecnología MDE para mejorar su reusabilidad, flexibilidad y extensibilidad. Una posible aplicación sería la arquitectura mostrada en la Fig. 4.4. Sin embargo, esta arquitectura es demasiado ambiciosa para el propósito de este trabajo, por lo que se tomarán algunos procedimientos centrados en facilitar la inclusión de nuevos sistemas P y bloques *hardware*, así como su depuración.

De ese modo, los modelos contemplados se muestran en la Fig. 4.5. El modelo tinyVHDL se trata de una representación de un diseño *hardware*, a nivel de interfaz entre sus componentes, y con detalles de una descripción VHDL del mismo, así como de la FPGA objetivo. Un diseño de una arquitectura contiene un conjunto de restricciones y parámetros, como es el modelo y familia de FPGA, y tiene una biblioteca asignada. Una biblioteca puede contener uno o más paquetes, en los que se definen cada uno de los componentes del sistema, describiendo la interfaz de cada uno a través de sus puertos de entrada, salida y entrada/salida, contemplando también aquellos parametrizados. Por último, se incluyen los elementos propios del lenguaje VHDL, la entidad y la arquitectura.

Una vez definido el modelo, se contempla una transformación, escrita empleando el lenguaje EGL de Epsilon, y enfocada a la generación del código *hardware*. En este sentido, se genera, de forma automática, la estructura de ficheros del proyecto, así como las interfaces de cada uno de los módulos y el $Makefile$ encargado de coordinar el proceso de síntesis y generación del fichero de programación de la FPGA.

Al igual que en la versión anterior, no se genera el código que describe el

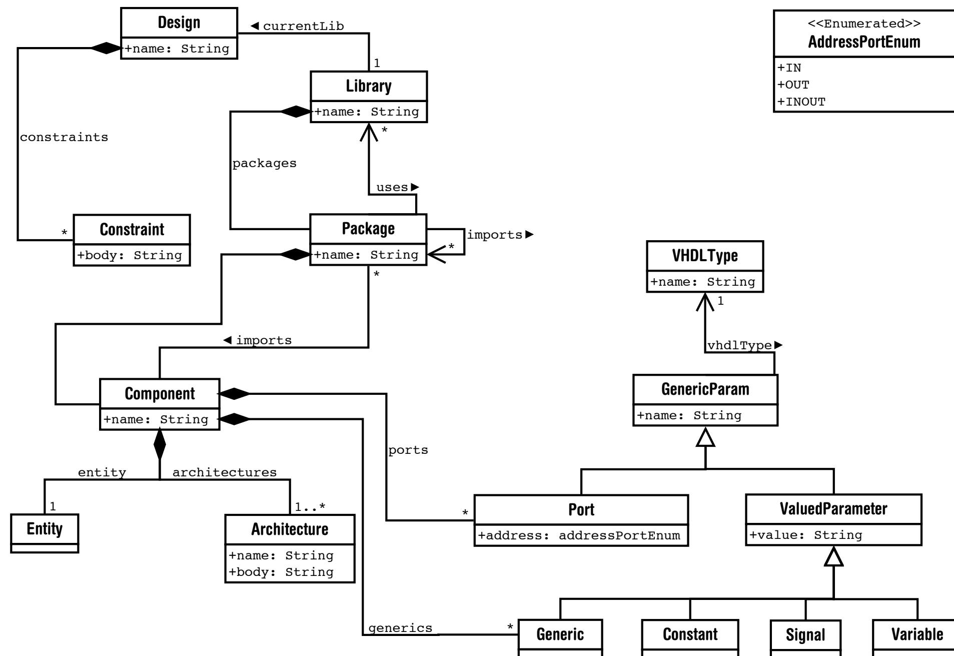


Fig. 4.5: Modelo tinyVHDL empleado en el *software*.

comportamiento funcional de los componentes. No obstante, el empleo del lenguaje EGL facilita la tarea de diseño, ya que, al estar basado en plantillas, permite escribir de forma directa el lenguaje objetivo, al mismo tiempo que se incorpora la sintaxis específica que aporta la reusabilidad del código.

En caso de que los componentes no sean de producción propia, esto es, *IP Core* del fabricante XILINX, las transformaciones EGL sí generan un código final, en base a las interfaces requeridas por la arquitectura objetivo. El código sintetizable es generado por la herramienta XILINX CORE GENERATOR SYSTEM, y al menos se precisan dos ficheros para sintetizar un *IP Core*. En caso de generar más de un *IP Core*, únicamente sería estrictamente necesario añadir un fichero .xco adicional por componente. Es importante destacar que, respecto a la generación en la primera versión del *software* generador, la cual era capaz de generar estos ficheros, este planteamiento se diferencia en la flexibilidad ofrecida, ya que permite fácilmente la extensión a otros modelos y familias, a diferencia del caso anterior, pensado para un modelo concreto de la familia VIRTEX7. Además, la modelización del sistema,

permite la generación conjunta de todos los *IP Core* necesarios haciendo uso del mismo fichero .xco, suponiendo un proceso más adecuado.

Existen dos ventajas adicionales, la primera de ellas es el hecho de ofrecer una lógica de generación más desarrollada. Así, para aquellos casos soportados, el sistema puede seleccionar, en base al análisis del modelo del sistema, la conversión más adecuada. Un ejemplo lo constituye la implementación del bloque de cómputo N_{r_i}, para el que existen dos posibles implementaciones. En este caso, es posible intercambiar cualesquiera de ellas de forma automática, según el modelo de sistema P. Esta funcionalidad se presenta como un compromiso futuro, ya que la extensión del trabajo únicamente ha permitido una implementación parcial de esta funcionalidad, orientada a componentes muy concretos.

La segunda ventaja está relacionada con mejoras en la metodología de desarrollo, especialmente en la realización de pruebas unitarias automatizadas. Este concepto se resume en la siguiente sección.

4.3.3. Aplicación de MDE en la generación de pruebas unitarias de módulos

Las pruebas unitarias son una necesidad para el mantenimiento, extensión y calidad de cualquier solución actual, independientemente de su naturaleza *software* o *hardware*. Por ello, durante el desarrollo de la arquitectura Almond PS, se ha desarrollado una prueba unitaria para cada módulo del sistema. En este aspecto, los propios lenguajes de descripción de *hardware*, así como la gran mayoría de los entornos de desarrollo, proporcionan sintaxis y entorno para la definición y realización de pruebas. Refiriéndonos al actual trabajo, el lenguaje empleado ha sido VHDL, junto con el *software* ISE Design Suite[5] del fabricante Xilinx, obteniendo herramientas destinadas a cubrir dos tipos de necesidades en este ámbito: la simulación *software* del diseño *hardware* y simulación *on chip* mediante una interfaz de comunicación con el chip FPGA que permite la captura, lectura e inyección de señales durante la ejecución del sistema implementado.

En relación al primero, se pueden definir pruebas unitarias empleando VHDL.

[5] ISE Design Suite: `http://www.xilinx.com/products/design-tools/ise-design-suite.html`

De ese modo, en combinación con un simulador es posible interpretar el código VHDL y realizar una simulación funcional del sistema. La suite de XILINX ofrece el simulador ISE SIMULATOR[6], que ha sido el empleado durante este trabajo. Este método presenta dos inconvenientes: la escasa capacidad de abstracción del leguaje empleado, y la limitada reutilización del código desarrollado, reducido a la ejecución de pruebas unitarias funcionales. Como se describe más adelante, gran parte de este trabajo puede automatizarse haciendo uso de la metodología MDE.

Con respecto al segundo, el fabricante XILINX no ofrece una herramienta de depuración completa, sino que se limita a proporcionar las herramientas necesarias para poder comunicarse con el chip FPGA, siendo responsabilidad del desarrollador la implementación del código y utilidades que permitan la generación de los patrones de test, la generación de los resultados esperados y el posterior contraste de estos últimos con los adquiridos. Como aspectos negativos, cabe destacar el esfuerzo necesario por parte del desarrollador, no únicamente en la parte de generación de patrones, resultados esperados y su verificación, sino también en la generación de los componentes *hardware* necesarios para la interfaz, así como en el tratamiento y generación de la información recibida y transmitida.

Desde hace tiempo, los entornos *software* facilitan enormemente la implantación de test unitarios a lo largo del desarrollo, persiguiendo la automatización y reutilización del código. Del mismo modo, existe un nutrido número de bibliotecas en los distintos lenguajes que complementan los servicios ofrecidos por las herramientas *IDE*. Algunos ejemplos los constituyen ECLIPSE de THE ECLIPSE FOUNDATION, o IntelliJ IDEA[7] de la compañía JETBRAINS como entornos de desarrollo, y los *framework* JUNIT[8] y UNITTEST[9] para los lenguajes de programación JAVA y PYTHON, respectivamente.

[6]ISE SIMULATOR: `http://www.xilinx.com/tools/isim.htm`
[7]https://www.jetbrains.com/idea/
[8]http://junit.org/
[9]https://docs.python.org/2/library/unittest.html

4.3.3.1. Estructura de una prueba unitaria haciendo uso de la metodología MDE

Tal y como se ha indicado en las capítulos anteriores, uno de los grandes retos de la implementación *hardware* de sistemas P es su gran dinamismo al ser un paradigma de computación orientado a máquinas, en la que las distintas variedades son desarrolladas por equipos independientes. En consecuencia, adicionalmente a un diseño del sistema adecuado, es preciso contar con herramientas y metodologías de desarrollo que faciliten, en la medida de lo posible, la reutilización del código y la colaboración entre equipos. En este sentido, se han realizado avances que persiguen la automatización de pruebas unitarias.

Antes de nada, es preciso detallar el grado de desarrollo del conjunto de herramientas. Estas se encuentran enfocadas a sistemas síncronos, donde en un único ciclo de reloj se activan[10] todas las señales de entrada, tanto de datos, como de control, y, tras una espera de n ciclos de reloj, el sistema devuelve la salida. Se contempla la necesidad de mantener las señales de entrada y salidas activas durante más de un ciclo, pero tanto su activación como su desactivación deben tener lugar en el mismo ciclo de reloj.

La realización de cada test comprende tres fases, según se muestra en la Fig. 4.6. La primera de ellas consiste en la generación del conjunto de señales (patrones) con las cuales se estimulará el componente *hardware*. La segunda comprende la estimulación del mismo, virtualmente para una simulación funcional o, físicamente para una simulación *on chip*, recogiendo los resultados arrojados por este. Finalmente, la tercera fase se encarga de comprobar que los resultados esperados coinciden con los realmente obtenidos en el paso anterior. De estas fases, la segunda debe ser manual, ya que no se ha desarrollado el componente *software* encargado de comunicarse con la aplicación servidor de las herramientas de XILINX empleadas. A pesar de ello, es posible realizar una simulación funcional y *on chip*, sin realizar ninguna modificación en la tercera fase.

Respecto a la simulación funcional, aunque la modelización del componente a testear se limite únicamente a su interfaz, es suficiente para automatizar gran parte

[10] Se entiende como señal activa cuando su contenido es válido. Por el contrario, se consideran señales inactivas cuando su contenido no es válido, reflejando un valor cualquier o manteniéndolas en alta impedancia.

de los procesos que implican los test unitarios. Así, en la primera fase es preciso generar los ficheros de simulación en VHDL, además del proyecto de *Testbench* para el simulador, al ser la segunda fase manual. De ese modo, el usuario debe iniciar este, abrir el proyecto generado e iniciar la simulación. En este caso, es el código generado el encargado de exportar los resultados obtenidos, y que deberán ser contrastados con los esperados en la tercera fase.

En caso de tratarse de una simulación *on chip*, el procedimiento es similar. Durante la primera fase es preciso generar directamente código sintetizable en la FPGA, además del fichero de proyecto para la herramienta. Del mismo modo, el usuario debe iniciar la aplicación, abrir el proyecto y llevar a cabo la adquisición de resultados. Como diferencia, en este caso los resultados deben ser exportados manualmente. Al hacer uso de las herramientas de XILINX, la última fase debe admitir sus ficheros de exportación, y la primera la generación de *IP Core* específicos.

Por último, es preciso comentar que actualmente no existe interfaz gráfica, por lo que tanto la ejecución como el informe de los resultados del test son ofrecidos por consola de comandos.

A continuación se describen de un modo más detallado cada una de las fases de testeo, con las herramientas desarrolladas haciendo uso de la metodología MDE.

4.3.3.2. Primera fase. Generación de entradas

En este fase es preciso generar el conjunto de valores de prueba, así como los elementos necesarios para el uso de las herramientas del paso 2. En primer lugar se detallan los aspectos comunes y posteriormente se comentan los detalles de cada tipo de simulación.

Respecto a los elementos necesarios, se generan (1) el código HDL que se desea probar, (2) la lógica necesaria para la activación de las señales, y (3) los elementos específicos de cada tipo de simulación, entre los que se incluyen los ficheros propios de las herramientas, comentados anteriormente. Estos ficheros son generados de forma automática a través de dos transformaciones, una de código a modelo, y otra de modelo a código. Para la primera transformación es necesario el análisis del código escrito por el usuario, en el que se incluyen algunos decoradores, para generar a partir de este el modelo del componente. A continuación, haciendo usos

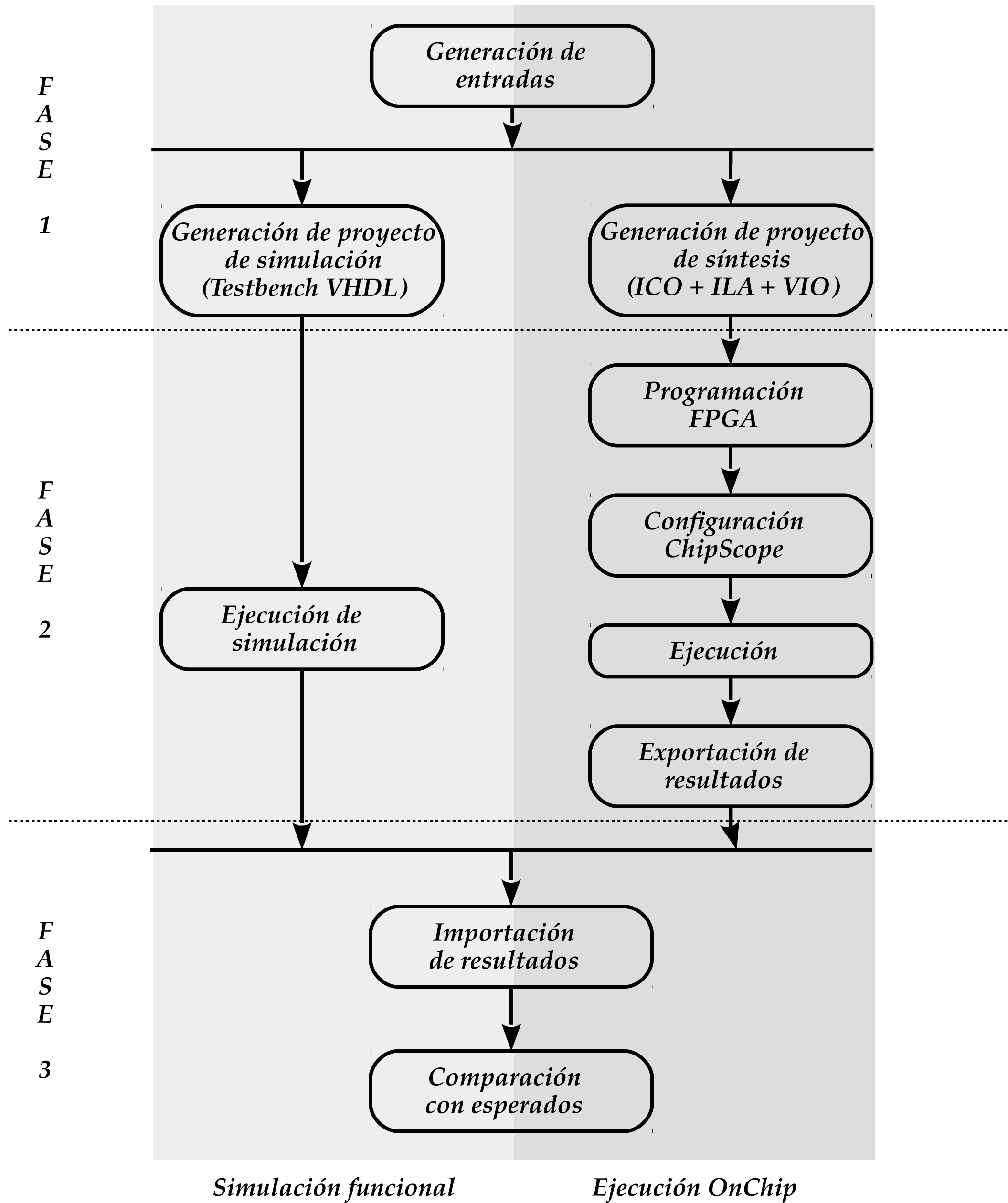


Fig. 4.6: Diagrama simplificado de los tres pasos de la metodología de pruebas.

de plantillas EGL, estos modelos son transformados al conjunto de fichero ficheros anteriormente enumerado.

De ese modo, la interfaz y parte de la funcionalidad es inferida directamente del modelo del componente, a lo que se suman algunos decoradores. Los decoradores actualmente soportados son $@l/h_clk$ para la señal de reloj, $@l/h_rst$ para la señal de *reset*, $@l/h_enable$ para la señal de activación, $@data$ para las señales de datos y $@ctrl$ para las señales de control; h/l hacen referencia al flanco en el que está activa la señal ($@l_clk$ y $@h_clk$, por ejemplo). Además de los anteriores, existen dos decoradores adicionales asociados al componente *hardware*. El primero de ellos es $@clk2out < numero >$, que indica el número de ciclos que tarda en ofrecer la salida tras recibir la entrada. El segundo es $@clk2init < numero >$, que indica el número de ciclos que requiere el componente para inicializarse tras recibir una señal de reset. En ambos casos, se considera por defecto el valor de 1.

Resta por definir el comportamiento funcional a testear. En este sentido, la automatización completa necesitaría modelar también el comportamiento funcional del componente, objetivo fuera del ámbito de este trabajo. Existen dos métodos alternativos. El primero de ellos consiste en escribir directamente un fichero en formato *Comma-Separated Values* (CSV), indicando la entrada y los resultados asociados a esta. La primera línea del fichero, el equivalente al título de las columnas, indica a qué señales hace referencia cada posición de cada línea. En caso de que el resultado sea irrelevante, se puede emplear un guión, $-$, o, sencillamente, dejar la posición en blanco.

El segundo método, consiste en emplear código JAVA para describir el comportamiento. Por lo tanto, es preciso el uso de una clase que se encargue de testear el componente. El enlace con el componente se realiza empleando un decorador $@java < url_clase >$.

Respecto a esta clase JAVA, se ofrece una herramienta que genera a su vez la clase con los métodos necesarios. Los más relevantes son $setUp()$, $tearDown()$, $initialize_dataTest$, $testComponent$, $getInput$ y run, que se describen brevemente a continuación.

- *setUp*. Método generado vacío. Debe contener la lógica de inicialización necesaria para cada test.

- *tearDown*. Método generado vacío. Debe contener la lógica de finalización necesaria para cada test.

- *initialize_dataTest*. Método generado vacío. Debe generar un objeto de tipo *java.util.List*, donde cada elemento es de tipo *java.util.Map<String, Object>*, en el que la clave del diccionario es el nombre de la señal, y el valor puede ser una cadena de texto (tipo *String*), un número entero (tipo *Integer*) o un valor lógico (tipo *Boolean*).

- *testComponent*. Método generado vacío. El método debe recibir como parámetro todas las señales que recibe el módulo, y devolver un elemento de clase $java.util.Map < String, Object >$, con las mismas restricciones que las detalladas en *initialize_dataTest*.

- *getInput*. Método autogenerado. A no ser que se quiera añadir nueva funcionalidad a la clase, no debe ser modificado. Se encarga de inicializar y finalizar las pruebas llamando a *setUp* y *tearDown* y generar los casos de prueba con *initialize_dataTest*.

- *run*. Método autogenerado. A no ser que se quiera añadir nueva funcionalidad a la clase, no debe ser modificado. Se encarga de inicializar y finalizar las pruebas llamando a *setUp* y *tearDown*, generar los casos de prueba con *initialize_dataTest*, y verificar el comportamiento con *testComponent*. Este método se corresponde con la tercera fase, se describe en esta sección para facilitar la comprensión del diseño, ya que se encuentra en la misma clase.

Aparte de los métodos mencionados anteriormente, existen algunos adicionales. Uno de los más relevantes, es el encargado de transformar las señales generadas por el método *initialize_dataTest*, ya que los datos son comparados empleando una representación de tipo *String*. Así, en caso de que el valor de la señal se represente con un *String*, se considera que es una salida literal del bus. En caso de que la anchura del bus sea diferente, adapta el tamaño al empleado en el componente. En caso de que el valor se represente con un *booleano*, se genera el valor "1" para representar el valor verdadero, y el "0" para representar el valor falso. Por último, el tipo entero se representa por defecto en complemento a dos.

La lógica necesaria para la generación del test *software* es la más sencilla de las dos. En este sentido, es preciso generar el código VHDL del test unitario, y los patrones de test, que se guardan en un fichero de texto en formato CSV, donde las columnas son separadas por punto y coma, y las filas por saltos de línea. Así, durante la simulación el código generado se limita a leer, de este fichero (*test bench*), las entradas con las que debe estimular al bloque *hardware* objetivo, y almacenar estas, junto con el resultado, en un nuevo fichero, manteniendo el formato CSV del anterior.

Respecto al test *hardware on chip*, este trabajo ha empleado las FPGA del fabricante XILINX, es por ello que únicamente se ofrece soporte para las herramientas de esta marca. En este sentido, las herramientas que ofrecen esta funcionalidad son CHIPSCOPE PRO y SERIAL I/O TOOLKIT. Ambas requieren el empleo de los *IP Core*: ILA, encargado de capturar las señales generadas en la FPGA; VIO, con el que se generan señales de entrada para el sistema, además de recoger la salida en tiempo real; y por último ICO, encargado de la comunicación con el *software*. Al tratarse de varios *IP Core*, el sistema genera los ficheros de parametrización necesarios, acompañados de los ficheros en VHDL para la construcción de un sistema que permita la estimulación del componente y la recogida de información, y el fichero $Makefile$ encargado de la generación del fichero *.bit*. Respecto a las señales de estímulo, estas se almacena directamente en *hardware*, o, en caso de ser necesario, en memorias BRAM. Por último, el sistema ofrece como salida una señal de disparo, con la que indicar a CHIPSCOPE cuando la salida es válida, con la intención de ahorrar recursos.

Es preciso destacar que es en esta fase donde la metodología MDE tiene mayor peso, al ofrecer herramientas encaminadas al análisis del código del desarrollador para la realización de un modelo del componente *hardware*, y su posterior tratamiento para la generación del conjunto de ficheros necesarios para la realización de las pruebas.

4.3.3.3. Segunda fase. Adquisición de resultados.

La siguiente fase consiste en la adquisición de los resultados. Tal y como se comentó anteriormente, esta fase no está automatizada, por lo que el desarrollador

debe seguir los procesos manualmente. En caso de la simulación *software*, consiste en ejecutar el proyecto en cualquiera de los simuladores disponibles, por ejemplo el ISE SIMULATOR (ISIM).

La simulación *hardware on chip* es similar, no obstante, el proceso es ligeramente más tedioso. En este caso, es necesario hacer uso del *software* CHIPSCOPE. Se debe programar la FPGA con el *.bit* generado por las herramientas, ejecutar la aplicación y realizar una configuración previa. En ella, se debe emplear la señal de disparo anteriormente mencionada como *trigger* de captura de resultados. Esto permite el almacenamiento de las muestras válidas, así como el ahorro de recursos, ya que los resultados son almacenados en componentes *hardware* de la FPGA, los bloques BRAM, y descargados una vez finalizado el proceso. Una vez adquiridos los resultados, no es necesario agrupar las señales, pero sí deben exportarse todas, en el mismo orden de captura. La exportación debe ser en formato *Tab-delimited ASCII format*.

4.3.3.4. Tercera fase. Contraste con resultados esperados.

Una vez obtenidos los resultados, es preciso volver al proyecto *software* generado en la primera fase, concretamente al método *testComponent*, detallado en primera fase. A diferencia de los métodos análogos *software*, este método se limita a comprobar si ambos valores, los esperados y adquiridos son iguales, sería el homólogo del método *assertEquals* de los *framework* de prueba *software*.

Un problema existente para los resultados de los test *hardware on chip*, es la sincronización de estos. En este caso, se contempla una búsqueda inicial del inicio de los resultados, al tiempo que se analiza el bit de disparo antes de efectuar la comparación, para el caso en el que los resultados no estén filtrados.

4.4. Conclusiones

En este capítulo se ha presentado, de forma breve, el desarrollo *software* asociado a la arquitectura Almond PS. Se ha presentado una primera versión que, aunque dedicada, contempla la flexibilidad requerida en la implementación de sistemas P, esto es, la modificación de componentes e, incluso, la inclusión de otros nuevos.

Así, se ha respetado la modularidad de la plataforma *hardware*, encapsulando la responsabilidad de generación en dos clases específicas.

En segundo lugar, se ha presentado una evolución de este desarrollo, especialmente enfocada a la flexibilidad y reusabilidad del código. Aunque este trabajo es demasiado extenso como para ser detallado en este documento, se han sentado los principios de diseño y los aspectos tecnológicos necesarios para la evolución del *software*, comprobando su utilidad y ventajas. Por otro lado, se ha desarrollado una metodología de pruebas unitarias que, aunque no se encuentra completamente automatizada, y existe aún un gran desarrollo por delante para ofrecer todos sus frutos, ha permitido acelerar el desarrollo de la arquitectura *hardware*, al tiempo que abre la puerta para su aplicación directa en otros ámbitos, trasladando las ventajas de este tipo de metodologías, con una larga trayectoria en el mundo del desarrollo *software* y con demostrada utilidad, al desarrollo *hardware*.

En consecuencia, se ha implementado un *software* que permite la generación de instancias concretas de la arquitectura Almond PS a partir de sistemas P de entrada, trasladando los principios de flexibilidad y reusabilidad a este ámbito. Además, este trabajo se ha extendido con las bases de una segunda iteración, donde priman aún más estos conceptos, y los inicios de métodos automáticos para facilitar la traslación de una metodología de desarrollo *software*, el desarrollo basado en pruebas, al mundo del desarrollo *hardware*.

Parte III

Escenarios de verificación, pruebas y resultados

Capítulo 5

Análisis de Almond PS. Pruebas y resultados

Una vez descrita la arquitectura Almond PS, en la Sección 5.1 se describen los sistemas de prueba a partir de los que se obtendrán los resultados. En la Sección 5.2 se presentan los resultados funcionales y se comparan con aquellos obtenidos con la herramienta P-Lingua. En la Sección 5.3 se analizan los recursos *hardware* consumidos por los sistemas P implementados y, finalmente, en la Sección 5.4 se concluye con un análisis de rendimiento.

5.1. Sistemas P de prueba

Antes de presentar el conjunto de resultados, se detallan los cuatro sistemas de prueba empleados. Así, en todos los resultados se han considerado cuatro sistemas de referencia. Estos sistemas cumplen las restricciones establecidas en la Sección 3.3 (Pág. 95), al considerar reglas cuyo grafo de dependencias forma una cadena, diferenciándose únicamente en la parte derecha de estas, es decir, en los objetos generados.

Considerando, para los cuatro sistemas de referencia, que el alfabeto de un sistema P con N reglas se define como $O = \{o_0, \ldots, o_N\}$, $N > 0$, y considerando la operación de suma como módulo $N + 1$, es posible describir estos cuatro sistemas P del siguiente modo:

- Sistema 1 (Circular)

$$r_i : \begin{cases} o_{i-1}o_i \rightarrow o_io_{i+1} & 1 \leq i < N, \\ o_{i-1}o_i \rightarrow o_0o_1 & i = N. \end{cases}$$

- Sistema 2 (Circular-2)

$$r_i : \begin{cases} o_{i-1}o_i \rightarrow o_{i+1}o_{i+2} & 1 \leq i \leq N, \end{cases}$$

- Sistema 3 (Lineal)

$$r_i : \begin{cases} o_{i-1}o_i \rightarrow o_io_{i+1} & 1 \leq i < N, \\ o_{i-1}o_i \rightarrow o_io_i & i = N. \end{cases}$$

- Sistema 4 (Opuesto), $1 \leq i \leq N$

$$r_i : \begin{cases} o_{i-1}o_i \rightarrow o_io_{i+1} & i \text{ mód } 2 = 1, \\ o_{i-1}o_i \rightarrow o_{i-2}o_{i-1} & \text{e.c.o.c.} \end{cases}$$

Así, para realizar las pruebas de la arquitectura desarrollada y obtener los resultados, se emplean los cuatro sistemas P de referencia, tomando distintos tamaños, medidos como número de reglas. De ese modo, se han considerado los tamaños de 10, 20, 30, 50, 70, 90, 110, 130, 150, 170 y 190reglas.

La elección de estos cuatro sistemas viene determinada por la predictibilidad de las soluciones generadas tras su ejecución, que facilita el análisis de los resultados funcionales. A continuación, en la Fig. 5.1 se muestra un ejemplo de cada una de las propuestas para un tamaño de 10 reglas.

Tal y como se observa en los sistemas con dependencia circular, cada regla repone uno de los objetos consumidos por la propia regla, a la vez que sustituye el objeto anterior por el siguiente, transmitiendo la multiplicidad hacia delante. De ese modo, cada objeto es generado y consumido en dos ocasiones, excepto los objetos extremos, que únicamente son afectados una vez. Como excepción, es necesario modificar la última regla para, a nivel global, no romper la relación circular.

Dependencia circular	Dependencia circular-2
$r_1 : o_0 o_1 \rightarrow o_1 o_2$	$r_1 : o_0 o_1 \rightarrow o_2 o_3$
$r_2 : o_1 o_2 \rightarrow o_2 o_3$	$r_2 : o_1 o_2 \rightarrow o_3 o_4$
$r_3 : o_2 o_3 \rightarrow o_3 o_4$	$r_3 : o_2 o_3 \rightarrow o_4 o_5$
$r_4 : o_3 o_4 \rightarrow o_4 o_5$	$r_4 : o_3 o_4 \rightarrow o_5 o_6$
$r_5 : o_4 o_5 \rightarrow o_5 o_6$	$r_5 : o_4 o_5 \rightarrow o_6 o_7$
$r_6 : o_5 o_6 \rightarrow o_6 o_7$	$r_6 : o_5 o_6 \rightarrow o_7 o_8$
$r_7 : o_6 o_7 \rightarrow o_7 o_8$	$r_7 : o_6 o_7 \rightarrow o_8 o_9$
$r_8 : o_7 o_8 \rightarrow o_8 o_9$	$r_8 : o_7 o_8 \rightarrow o_9 o_{10}$
$r_9 : o_8 o_9 \rightarrow o_9 o_{10}$	$r_9 : o_8 o_9 \rightarrow o_{10} o_0$
$r_{10} : o_9 o_{10} \rightarrow o_0 o_1$	$r_{10} : o_9 o_{10} \rightarrow o_0 o_1$

Dependencia lineal	Dependencia opuesta
$r_1 : o_0 o_1 \rightarrow o_1 o_2$	$r_1 : o_0 o_1 \rightarrow o_1 o_2$
$r_2 : o_1 o_2 \rightarrow o_2 o_3$	$r_2 : o_1 o_2 \rightarrow o_0 o_1$
$r_3 : o_2 o_3 \rightarrow o_3 o_4$	$r_3 : o_2 o_3 \rightarrow o_3 o_4$
$r_4 : o_3 o_4 \rightarrow o_4 o_5$	$r_4 : o_3 o_4 \rightarrow o_2 o_3$
$r_5 : o_4 o_5 \rightarrow o_5 o_6$	$r_5 : o_4 o_5 \rightarrow o_5 o_6$
$r_6 : o_5 o_6 \rightarrow o_6 o_7$	$r_6 : o_5 o_6 \rightarrow o_4 o_5$
$r_7 : o_6 o_7 \rightarrow o_7 o_8$	$r_7 : o_6 o_7 \rightarrow o_7 o_8$
$r_8 : o_7 o_8 \rightarrow o_8 o_9$	$r_8 : o_7 o_8 \rightarrow o_6 o_7$
$r_9 : o_8 o_9 \rightarrow o_9 o_{10}$	$r_9 : o_8 o_9 \rightarrow o_9 o_{10}$
$r_{10} : o_9 o_{10} \rightarrow o_{10} o_{10}$	$r_{10} : o_9 o_{10} \rightarrow o_8 o_9$

Fig. 5.1: Ejemplo de sistemas P de referencia para un tamaño de 10 reglas.

El resultado esperado es una distribución homogénea de las multiplicidades de los objetos.

Atendiendo a la dependencia circular-2, esta es similar a la anterior. Cada regla consume dos elementos y genera los dos inmediatamente siguientes, trasladando la multiplicidad hacia los objetos de mayor índice. No obstante, en esta ocasión la última regla mantiene el esquema del resto, por lo que no se consigue una condición de equilibrio similar a la alcanzada anteriormente. Así, cada objeto es consumido y generado en dos ocasiones, a excepción de los extremos: el objeto de índice 0 tiene una relación generados/consumidos de 2/1, mientras que el de índice 1 de 1/2, y el de mayor multiplicidad, 10 en el caso del ejemplo, 2/1. Esto transforma los objetos de mayor y menor índice en acumuladores de la multiplicidad existente en el sistema. Es por ello que los sistemas alcanzan una condición de parada, motivada por la desaparición de objetos de índice 1, que estrangula la propagación de multiplicidades. Tras romper el círculo de propagación, el sistema ejecuta algunas reglas más, acumulando las multiplicidades en los objetos de índice extremo, hasta que el agotamiento del objeto anterior al de mayor índice, 9 en el caso del ejemplo, vuelve a romper la dependencia, ahora lineal.

Respecto a la dependencia lineal, es muy similar a la circular, diferenciándose únicamente en cómo la última regla modifica la dependencia global del sistema. En este caso, se emplea como acumulador el objeto de mayor índice, 10 en el ejemplo. Así, todos los objetos presentan un valor de generados/consumidos de 2/2, a excepción del objeto de índice 0, 0/1, del 1, 1/2, y el de mayor índice, 10 en el ejemplo, con una relación 3/1. Esto hace que el sistema alcance la condición de parada de forma rápida, acumulando las multiplicidades en el objeto de mayor índice. En este caso, la relación de propagación está claramente limitada por el déficit en la generación de los objetos de menor índice, 0 y 1, mientras que por el extremo superior, la mayor relación de generación del objeto de mayor índice, hace que sea el inmediatamente anterior, índice 9 en el ejemplo, el que rompa la propagación y se agote en todas las ejecuciones.

Por último, la dependencia opuesta es la más particular de las utilizadas. El objetivo es que el sistema no alcance nunca la condición de parada, empleando los objetos de índice impar como elementos de control, que evitan romper la continua ejecución del sistema. Así, estos objetos presentan siempre una relación

generados/consumidos de 1/1, siendo el elemento por el que compiten las reglas. De ese modo, y en alusión al ejemplo, las reglas r_5 y r_6 compiten por el objeto o_5, condicionadas siempre a que existan suficientes objetos o_4 y o_6, respectivamente, constituyendo una pareja de competencia. Las multiplicidades son siempre repuestas por la regla ganadora, esto es, r_5 repone los objetos de r_6 y viceversa. Adicionalmente, para cada pareja, la multiplicidad puede ser transferida a las parejas vecinas. Así, en el caso de la pareja formada por las reglas r_5 y r_6, los objetos extremos, o_4 y o_6 son también consumidos/generados por las parejas formadas por las reglas r_3 y r_4, y r_7 y r_8, respectivamente. Recuérdese que estos son restituidos en posteriores transiciones, por lo que el sistema no alcanza nunca la condición de parada. El resultado esperado muestra una multiplicidad de 1 unidad para los objetos de índice impar, mientras que aquellos de índice par actúan como acumuladores parciales, con una multiplicidad que fluctúa en un valor medio de 1 unidad, pudiendo alcanzar un rango variable de valores.

Atendiendo a la configuración inicial, en las pruebas funcionales se ha considerado el peor caso, aquel en el que existe competencia entre todas las reglas, por lo que se dota a cada objeto de una multiplicidad inicial de 1 unidad. Tomando como base estos resultados, el comportamiento se verifica del mismo modo para casos habituales, donde no existe una competencia tan acusada.

Respecto a las pruebas no funcionales, la configuración inicial concreta del sistema pierde relevancia, a favor de los recursos empleados por la codificación de las multiplicidades de los objetos. Así, lo importante es el tamaño en bits reservado para esta multiplicidad, que será modificado según los resultados a generar. En consecuencia, la multiplicidad inicial de todos los objetos se corresponde con el menor valor que hace uso de todos los bits disponibles. No obstante, podría haberse elegido cualquier otro valor representable.

Por último, se ha establecido, para todos los resultados obtenidos, un límite de 8192 transiciones para cada ejecución. Así, el sistema alcanza una condición de parada total si converge hacia una configuración a partir de la cual el conjunto de aplicabilidad es vacío. En caso contrario, alcanza una condición de parada de límite 8192 transiciones[1]. Para cada resultado, se ha tomado una muestra de 1024 ejecuciones diferentes. La diferencia entre cada una de estas ejecuciones se corres-

[1]En la Sección 2.1.1, Pág. 37, se describen con mayor detalle las condiciones de parada.

ponde con la inicialización del generador de números aleatorios, que le confiere al sistema su comportamiento no determinista.

5.2. Resultados funcionales

En esta sección se analizan los resultados funcionales de la arquitectura Almond PS. Se encuentra dividido en tres apartados, donde se analizan las configuraciones finales, el número de transiciones para alcanzar una configuración de parada, y la distribución, en el espacio de soluciones, de las configuraciones finales obtenidas. En cada uno de los apartados se han llevado a cabo dos baterías de resultados experimentales, tomando, en cada una de ellas, los cuatro sistemas de referencia. En la primera, se han obtenido los resultados referentes a la arquitectura Almond PS. En la segunda, se han obtenido los resultados usando P-Lingua.P-Lingua es una implementación *software* madura, flexible y bien respaldada. Es ampliamente utilizada para el diseño de nuevos sistemas y la confrontación de resultados de nuevas implementaciones y sistemas desarrollados. Por todo ello, contrastaremos los resultados obtenidos por Almond PS con los obtenidos por P-Lingua, que será considerada una implementación de referencia.

Atendiendo a los sistemas de prueba, se han considerado los descritos en el apartado anterior. No obstante, en relación a la representación de los resultados obtenidos, estos únicamente se muestran para un tamaño de reglas de 20 unidades (suficientemente pequeño como para poder ser bien representado a una escala conveniente en formato A4, y a la vez lo suficientemente grande como para que se pueda verificar el comportamiento de la arquitectura Almond PS). Para el resto de tamaños, se analiza el comportamiento funcional, mostrando gráficamente los resultados más relevantes.

5.2.1. Análisis de las configuraciones finales

En primer lugar, se analiza la validez de los resultados obtenidos, atendiendo, exclusivamente, al valor de las multiplicidades de los objetos en las configuraciones finales. Así, se estudia el comportamiento de la implementación *hardware*, tomando en consideración la definición de los sistemas de referencia, para posteriormente

contrastarlo con la implementación *software* P-Lingua.

5.2.1.1. Análisis de Almond PS

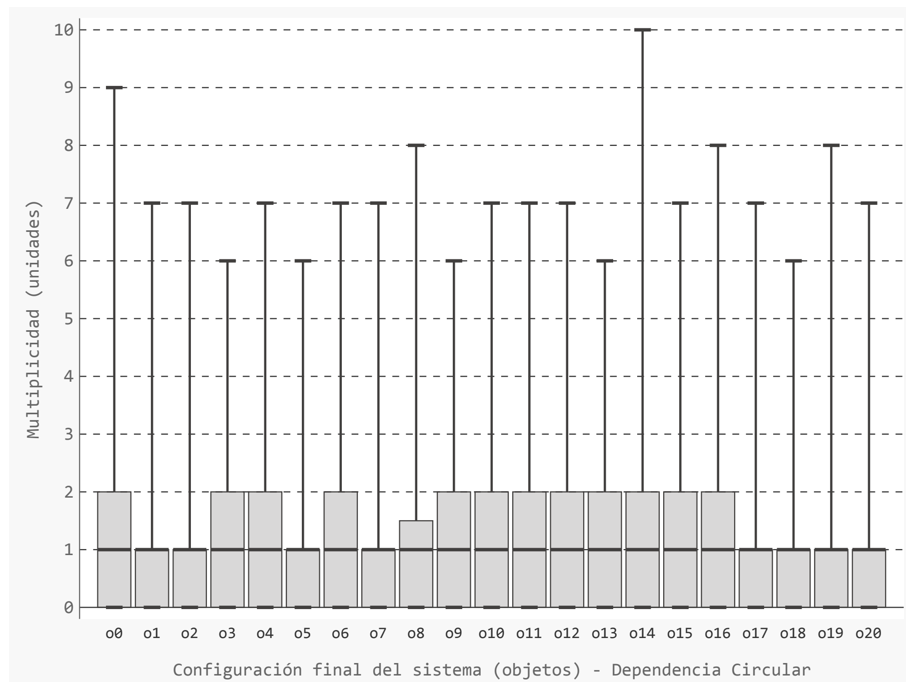
A continuación, se analiza si la arquitectura Almond PS ofrece los resultados esperados, según las definiciones de cada uno de los modos de dependencia presentados, estudiando cada uno de estos de forma independiente.

Dependencia circular

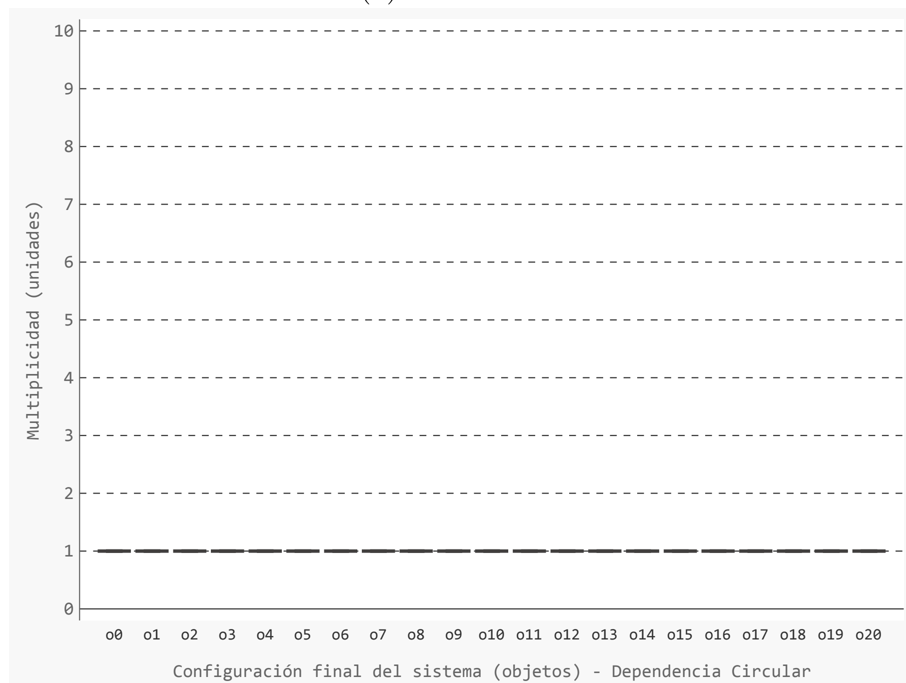
Atendiendo a la dependencia circular, la Fig. 5.2 muestra la configuración final de un sistema P con un tamaño de 20 reglas. Respecto a la condición de parada, alcanza una condición de parada límite de 8192 transiciones. Se corresponde con una diagrama de cajas y bigotes en el que se representa, para cada objeto, los cuartiles $Q0$, $Q1$, $Q2$, $Q3$ y $Q4$ de la distribución seguida por las multiplicidades de los objetos. La figura superior (a) se corresponde con los resultados experimentales obtenidos, mientras que la inferior (b) hace referencia a los resultados esperados o más probables, según la definición de este tipo de sistemas. En este sentido, se comprueba como los resultados se distribuyen alrededor de la unidad, el valor más probable según la naturaleza del sistema y siguiendo, en consecuencia, un comportamiento similar al esperado.

Como resultado extraño a priori, el valor obtenido para el objeto o_8, un número no entero, es correcto. Esta cifra se debe a que el tamaño de la muestra es par, por lo que para el cálculo de los cuartiles se emplea la media de los dos valores más próximos.

En la Fig. 5.3 se representa en (a) el valor medio y máximo de las distancias, respecto al comportamiento más probable, de los sistemas de tamaño 20 reglas, y en (b) los valores máximos y medios de las medias de las distancias de todos los tamaños considerados (10 a 190 reglas). En este sentido, las distancias obtenidas en (a) respaldan los anteriores diagramas, y en (b) se extiende el resultado a los distintos tamaños considerados en los sistemas prueba. Así, al considerar todos los tamaños (b), se comprueba como la distancia media siempre es inferior a la unidad, lo que indica que el grueso de las multiplicidades fluctúa entre los valores 0, 1 y 2, esto es, alrededor de la unidad, el valor más probable. Los valores máximos son fruto de la naturaleza no determinista de este modelo de computación, y por sí

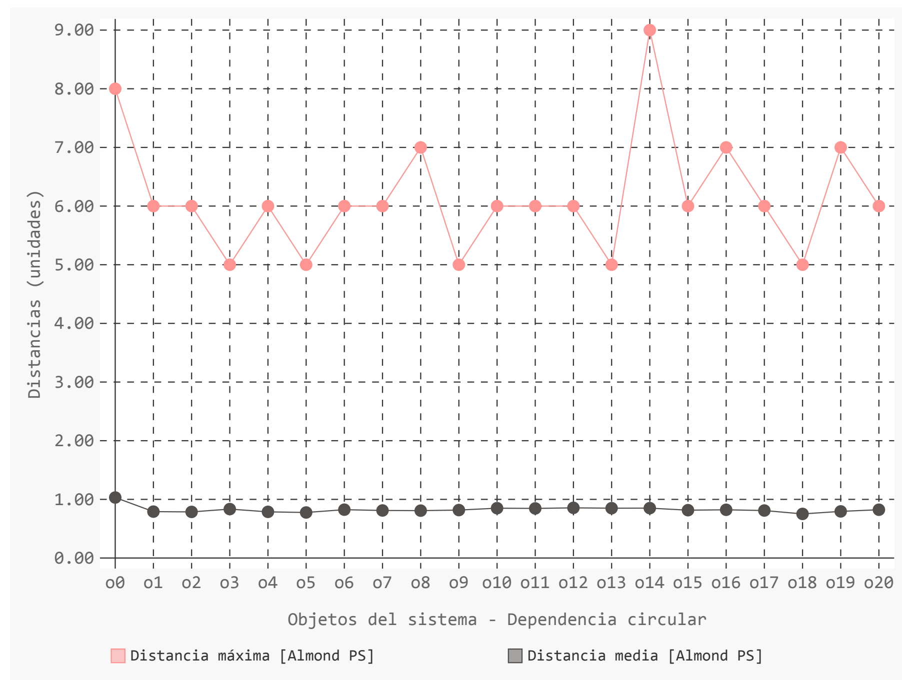


(a) Almond PS

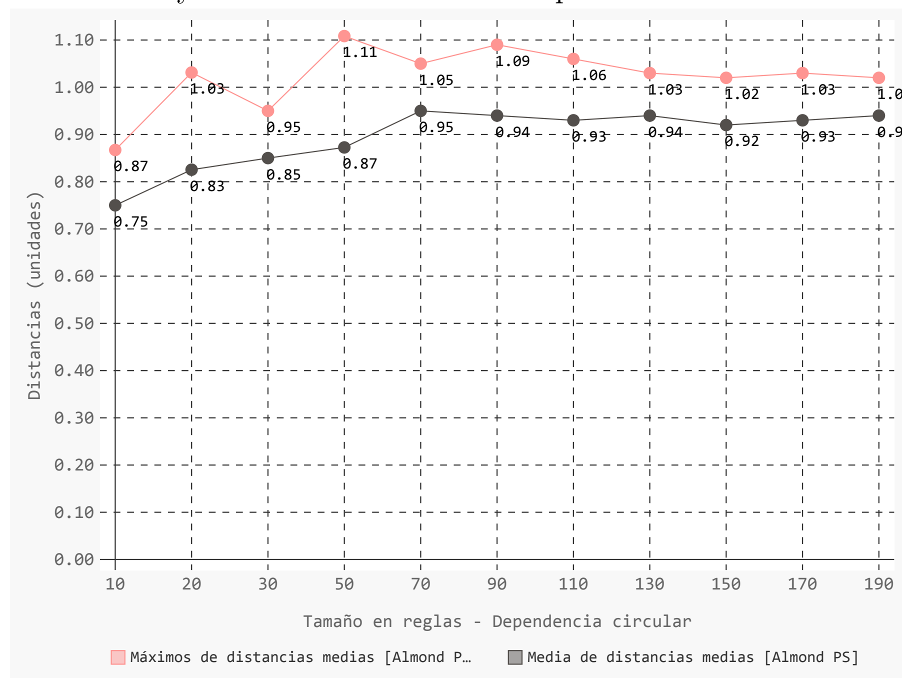


(b) Resultados más probables según definición del sistema

Fig. 5.2: Diagrama de cajas y bigotes que muestra los cuartiles $Q0$ (valor mínimo), $Q1$, $Q2$ (mediana), $Q3$ y $Q4$ (valor máximo) de la distribución seguida por las multiplicidades de los objetos tras la ejecución de los sistemas P con dependencia circular. En (a) se representan los resultados obtenidos con Almond PS y en (b) una aproximación de los más probables según la definición del sistema.



(a) Valor medio y máximo de las distancias para un sistema P de 20 reglas



(b) Valor medio y máximo de las medias de las distancias de sistemas P de distinto tamaño

Fig. 5.3: Las dos figuras muestran la diferencia de los resultados obtenidos con la plataforma Almond PS, respecto a los resultados más probables para sistemas P con dependencia circular. En la gráfica superior se representa el valor máximo y la media de las distancias de las multiplicidades de los objetos tras la ejecución de los sistemas P con un tamaño de 20 reglas. La gráfica inferior representa el valor máximo y medio de las medias de las distancias de las multiplicidades para los sistemas P de tamaño 10, 20, 30, 50, 70, 90, 110, 130, 150, 170 y 190.

solos no representan un resultado significativo. No obstante, para el tamaño de 20 reglas (a) se puede comprobar como se encuentra comprendido entre los valores 5 y 7, con dos excepciones. Para la extensión a todos los tamaños (b) los máximos de los distancias medias superan ligeramente la unidad. Para esta representación, se observa como los valores medios y máximos tienden a converger hacia un valor ligeramente inferior a la unidad, conforme los sistemas aumentan de tamaño, tras unos valores inestables para tamaños inferiores a 70 reglas.

Dependencia circular-2

En relación a la dependencia circular-2, el comportamiento esperado es aquel que conduce a la acumulación de las multiplicidades en los objetos extremos de la cadena, objetos primero, o_0, y último, o_N, presentando sus adyacentes, objetos segundo, o_1, y penúltimo, o_{N-1}, una multiplicidad de valor 0. En este sentido, es esperable que la multiplicidad acumulada por el primer objeto sea mayor que la del último. Este comportamiento se observa en la Fig. 5.4, para un sistema P de tamaño de 20 reglas. Al igual que para la dependencia circular, la representación se trata de un diagrama de cajas y bigotes, donde se representan los cuartiles, mediana incluida, para ofrecer información acerca de la distribución de las multiplicidades para cada objeto. Se observa como el sistema presenta un comportamiento similar al más probable. Así, o_0 recoge la multiplicidad mayor del sistema, seguido por o_n. Entre ambos concentran la práctica totalidad de las multiplicidades. También se observa como o_1 y o_{N-1} tienen, en todos los casos, una multiplicidad nula. En relación a los otros objetos, en las configuraciones finales sus multiplicidades están establecidas, en la mayoría de los casos, en 0 unidades, tomando en ocasiones el valor 1 y, dado su carácter no determinista, alcanzado valores atípicos de hasta 6 unidades.

En la Fig. 5.5 se extienden los resultados tomados para el tamaño de 20 reglas a todos los tamaños considerados en los sistemas de prueba. Como representación, se ha escogido un gráfico de líneas, que representa la multiplicidad media contextualizada de las medias de todos los objetos de los distintos tamaños de sistemas. Para su cálculo se ha dividido los valores de multiplicidad media entre la multiplicidad total presente en el sistema. Esta representación permite visualizar en una única gráfica resultados de distintos tamaños de sistemas (contextualizados a este

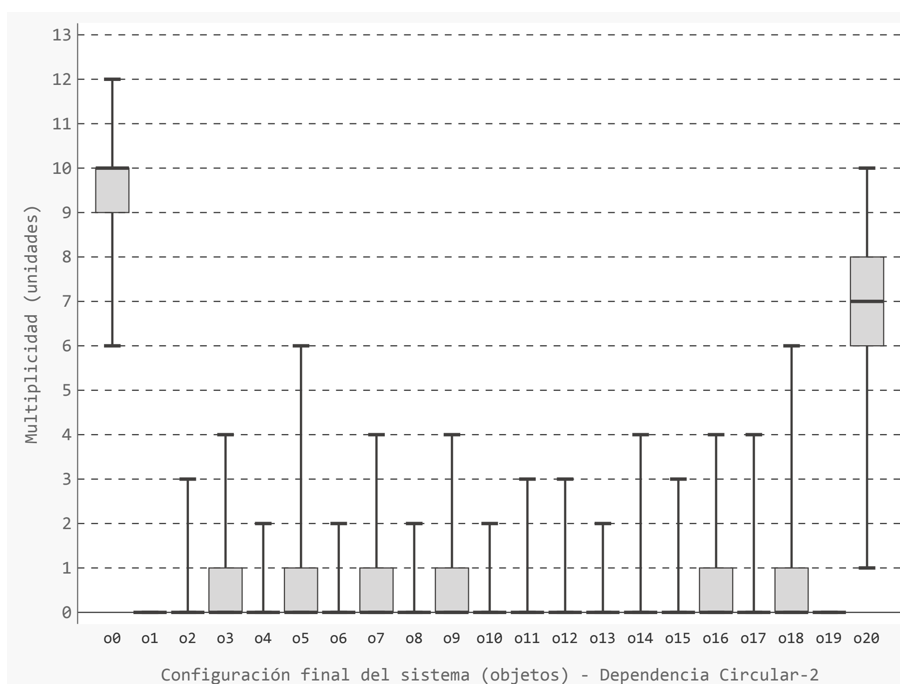


Fig. 5.4: Diagrama de cajas y bigotes que muestra los cuartiles $Q0$ (valor mínimo), $Q1$, $Q2$ (mediana), $Q3$ y $Q4$ (valor máximo) de la distribución seguida por las multiplicidades de los objetos tras la ejecución de los sistemas P con dependencia circular-2.

valor), al tiempo que evita su distorsión, ya que no se crean ni distruyen multiplicidades, únicamente se transmiten entre los distintos objetos. Para la elaboración de los resultados adquiridos, se han considerado 4 grupos de objetos, en base al comportamiento esperado. El primer y segundo grupo están compuestos por un único objeto, o_0 y o_N, respectivamente. El resultado más probable, al igual que para los sistemas de 20 reglas, es que ambos concentren la práctica totalidad de las multiplicidades, siendo el valor de o_0 superior a o_N. El tercer grupo, contiene 2 objetos, o_1 y o_{N-1}. La multiplicidad final de ambos deber ser nula, por lo que este grupo debe presentar un valor 0 en todos los casos. Por último, el cuarto grupo engloba al resto de objetos, desde o_2 hasta o_{N-2}, ambos inclusive. Estos objetos deben presentar una multiplicidad cercana a 0. Los resultados mostrados por la gráfica se ajustan a los descritos anteriormente. Así, el tercer grupo (o_1 y o_{N-1}), toma en todos los casos un valor nulo, y el cuarto grupo (resto de objetos), muestra un valor prácticamente igual a 0. Respecto a los dos primeros grupos, se observa

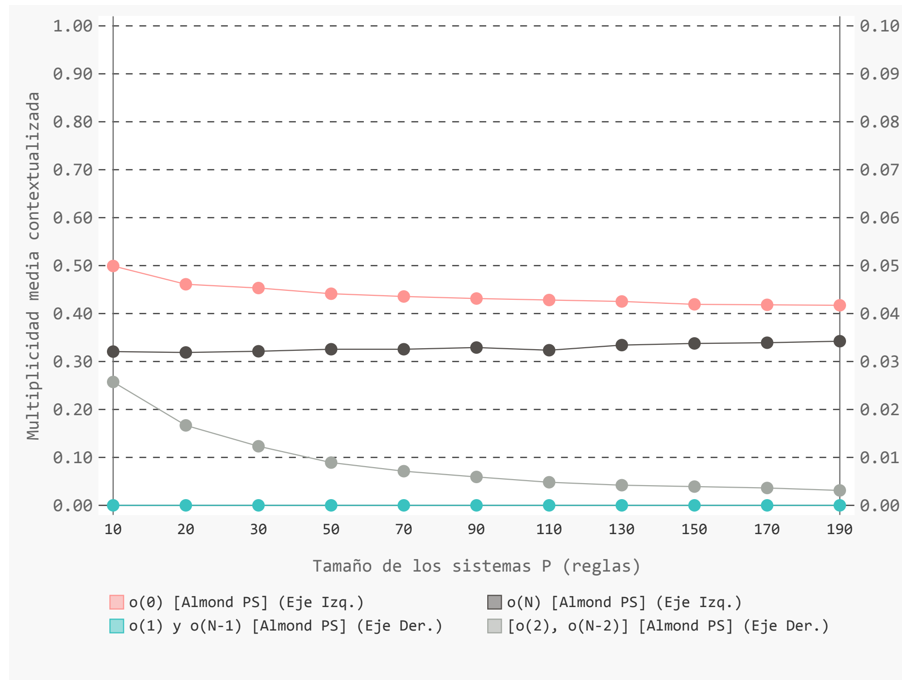


Fig. 5.5: Multiplicidad media contextualizada (multiplicidad media dividida entre la multiplicidad total presente en el sistema) de las medias de las multiplicidades de los objetos de los sistemas P, con tamaños comprendidos entre 10 y 190 reglas y dependencia circular-2, tras su ejecución bajo la implementación Almond PS. Para cada tamaño se muestra la multiplicidad para los objetos más característicos: los extremos (o_0 y o_N) y sus adyacentes (o_1 y o_{N-1}), así como el resto de objetos agrupados.

como concentran la mayoría de las multiplicidades, siendo el valor de o_0 siempre mayor al de o_N.

Dependencia lineal

Respecto a la dependencia lineal, presenta un único acumulador localizado en el último objeto de la cadena, al no existir un movimiento circular de multiplicidades. En la Fig. 5.6, se muestra la distribución de las multiplicidades de cada uno de sus objetos para un sistema P con un tamaño de 20 reglas, bajo los sistemas de prueba considerados. De ese modo, su comportamiento esperado es el de presentar en el último objeto la mayoría de las multiplicidades, agotando las del penúltimo objeto. A priori, el resto de los objetos deben presentar una multipli-

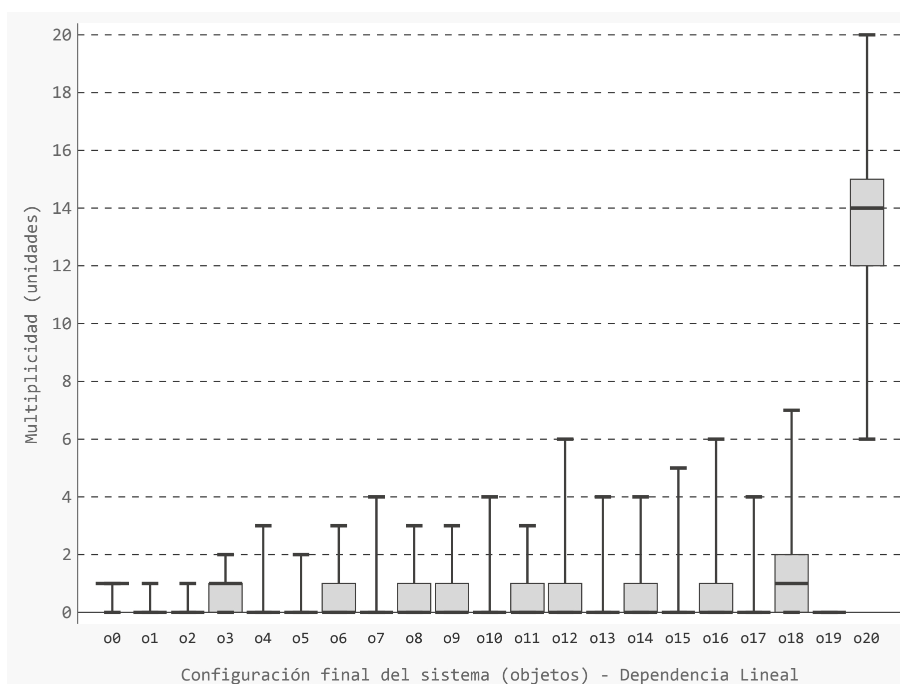


Fig. 5.6: Diagrama de cajas y bigotes que muestra los cuartiles $Q0$ (valor mínimo), $Q1$, $Q2$ (mediana), $Q3$ y $Q4$ (valor máximo) de la distribución seguida por las multiplicidades de los objetos tras la ejecución de los sistemas P con dependencia lineal.

cidad nula. No obstante, el hecho de que no exista retroalimentación, al tratarse de una dependencia lineal, hace que sean frecuentes los acumuladores parciales. Es por ello que existen objetos que presentan una multiplicidad igual a 1 unidad, presentando algunos valores atípicos mayores. Otro comportamiento apreciable, es la multiplicidad del objeto de índice menor, o_0. Este objeto no es generado por ninguna regla, y es consumido únicamente por r_1 $(o_0, o_1 \rightarrow o_1 o_2)$, en competencia directa con r_2 $(o_1, o_2 \rightarrow o_2 o_3)$. Al iniciar cada objeto con una multiplicidad igual a 1, si en la primera transición se aplica r_1, su multiplicidad será nula. En caso contrario, conserva su multiplicidad al consumir r_2 el objeto o_1, necesario para la ejecución de r_1. Por último, el antepenúltimo objeto, o_{18} en la Fig. 5.6, presenta una multiplicidad ligeramente superior al resto. Esto es debido al agotamiento del penúltimo objeto, o_{19}, que no permite la aplicación de la última regla.

La Fig. 5.7 muestra como los resultados mostrados para el tamaño de 20 reglas, se extienden al resto de tamaños considerados. La representación gráfica elegida ha

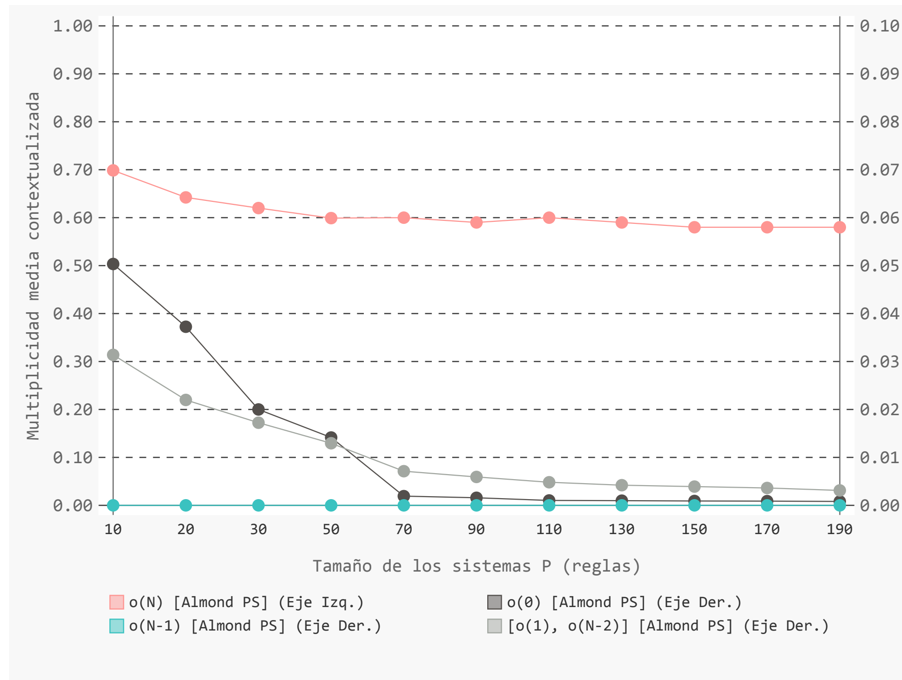


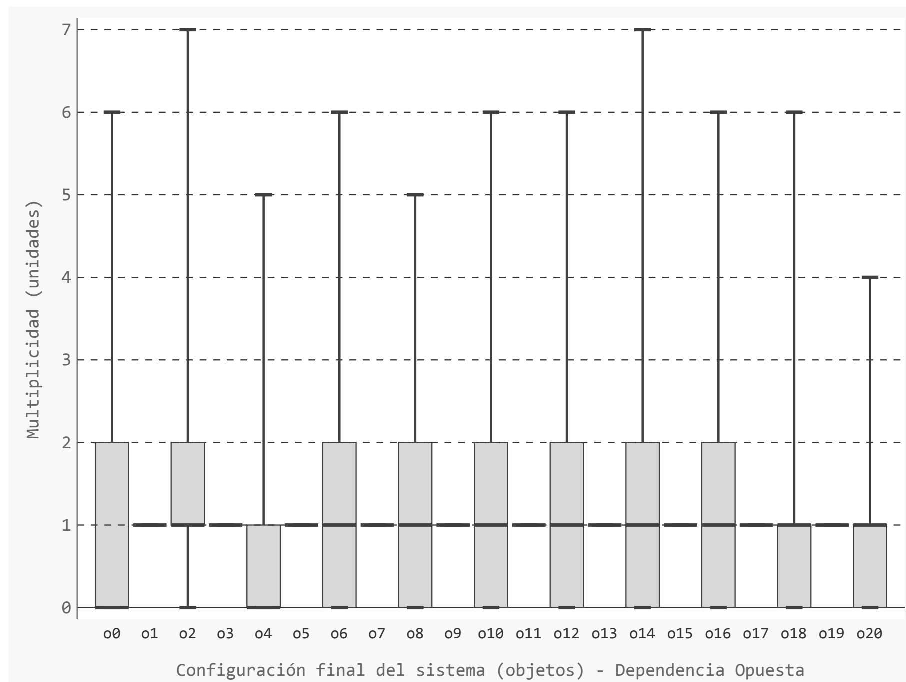
Fig. 5.7: Multiplicidad media contextualizada (multiplicidad media dividida entre la multiplicidad total presente en el sistema) de las medias de las multiplicidades de los objetos de los sistemas P, con tamaños comprendidos entre 10 y 190 reglas y dependencia lineal, tras su ejecución bajo la implementación Almond PS. Para cada tamaño se muestra la multiplicidad para los objetos más característicos: los extremos (o_0 y o_N) y o_{N-1}, así como el resto de objetos agrupados.

sido un gráfico de líneas, y se ha efectuado una división del conjunto de objetos en cuatro grupos, atendiendo a las características más relevantes de la dependencia analizada. El primer y segundo grupo comprenden los objetos o_0 y o_N, respectivamente. En referencia al objeto o_0, debe mostrar una media contextualizada cercana a 0, ya que la ejecución persigue el agotamiento de este objeto, entre otros. En los resultados obtenidos, se aprecia como la significancia del valor va decrementándose hasta confundirse con un valor nulo para tamaños superiores a 70 reglas. Este comportamiento coincide con el esperado, ya que este objeto finaliza con una multiplicidad igual a 0 ó 1, por las razones detalladas en el anterior párrafo, estando su media comprendida entre estos dos valores. Al aumentar el tamaño del sistema, también lo hace la suma de las multiplicidades de sus objetos, ya que hay que recordar que en la configuración inicial todos los objetos presentan una multiplici-

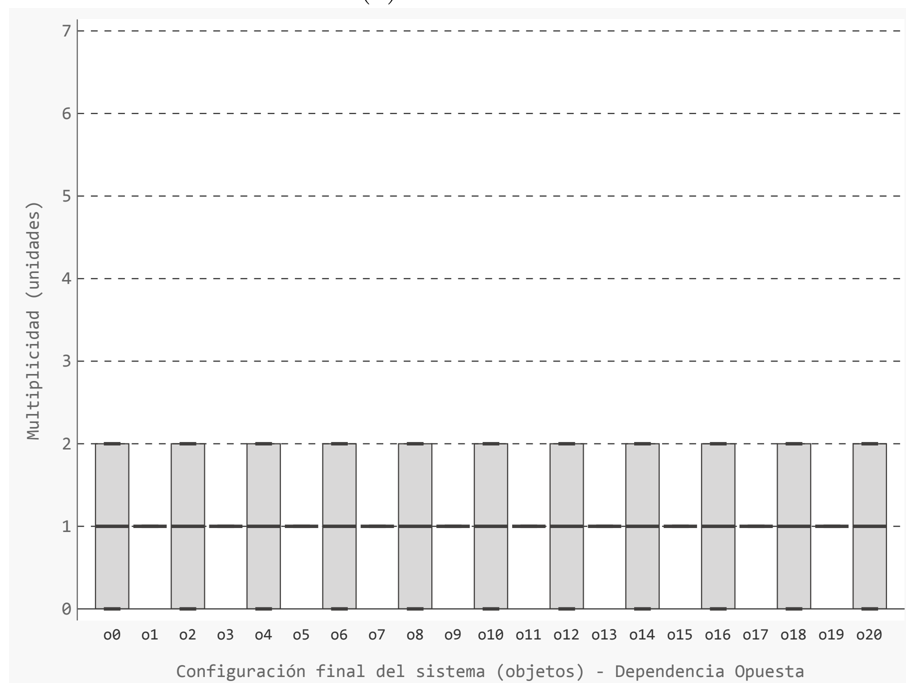
dad igual a la unidad, y durante la ejecución estas multiplicidades se transmiten a través de los objetos. Este hecho hace que se decremente la relevancia del objeto a, prácticamente, el valor nulo. En relación al objeto o_N, se aprecia como concentra la gran mayoría de la multiplicidad total del sistema, estabilizándose en un valor contextualizado de 0,6. El tercer grupo comprende el objeto o_{N-1}. La multiplicidad de este objeto debe ser 0 en todas las ejecuciones, para todos los tamaños, tal y como se aprecia en la gráfica. Por último, el cuarto grupo contiene el resto de los objetos, esto es, desde o_1 hasta o_{N-2}. Durante la ejecución, las multiplicidades son transmitidas por todos ellos hasta llegar a o_N. Por lo tanto, deben presentar un valor cercano a 0, tal y como se observa en la figura.

Dependencia opuesta

La dependencia opuesta es la última analizada. La Fig. 5.8 muestra los resultados obtenidos tras la ejecución de los sistemas P de tamaño 20 reglas, considerando este el sistema de prueba. En (a) se representan los valores obtenidos tras la ejecución con la plataforma Almond PS, y en (b) los valores más probables según las características de la dependencia analizada. En este sentido, su diseño persigue la no convergencia del sistema, sin presentar una dependencia circular en la cadena. Así, no existe ninguna regla que traslade, de forma directa, la multiplicidad de los objetos de mayor índice a los de menor índice, como sí ocurre en las dependencias circular y circular-2. En este caso, los objetos de índice impar actúan como acumuladores de control, siendo consumidos y generados en la misma proporción por cada una de las reglas, que se agrupan por parejas. Las reglas de una pareja compiten por estos objetos de control, y cada una de ellas compite, a su vez, con las parejas adyacentes, por el otro objeto restante. Por ejemplo, en la pareja formada por $r_3 : o_2, o_3 \rightarrow o_3 o_4$ y $r_4 : o_3, o_4 \rightarrow o_2 o_3$, el objeto o_3 es el de control, así, r_3 compite con la pareja de la izquierda, ($r_1 : o_0, o_1 \rightarrow o_1 o_2$, $r_2 : o_1, o_2 \rightarrow o_0 o_1$), por el objeto o_2, y r_4 compite con la pareja de la derecha, ($r_5 : o_4, o_5 \rightarrow o_5 o_6$, $r_6 : o_5, o_6 \rightarrow o_4 o_5$), por el objeto o_4. De esta forma, los objetos de control siempre deben mostrar una multiplicidad igual a la unidad, mientras que los restantes, deben presentar una multiplicidad comprendida entre 0 y 2. Este comportamiento se observa en la Fig. 5.8, donde los 10 objetos de control, aquellos de índice impar, presentan siempre una multiplicidad igual a la unidad, y 7 de los 11 objetos res-



(a) Almond PS



(b) Resultados teóricos (aproximación de valores esperados)

Fig. 5.8: Diagrama de cajas y bigotes que muestra los cuartiles Q0 (valor mínimo), Q1, Q2 (mediana), Q3 y Q4 (valor máximo) de la distribución seguida por las multiplicidades de los objetos tras la ejecución de los sistemas P con dependencia opuesta. En (a) se representan los resultados obtenidos con Almond PS y en (b) una aproximación de los más probables según la definición del sistema.

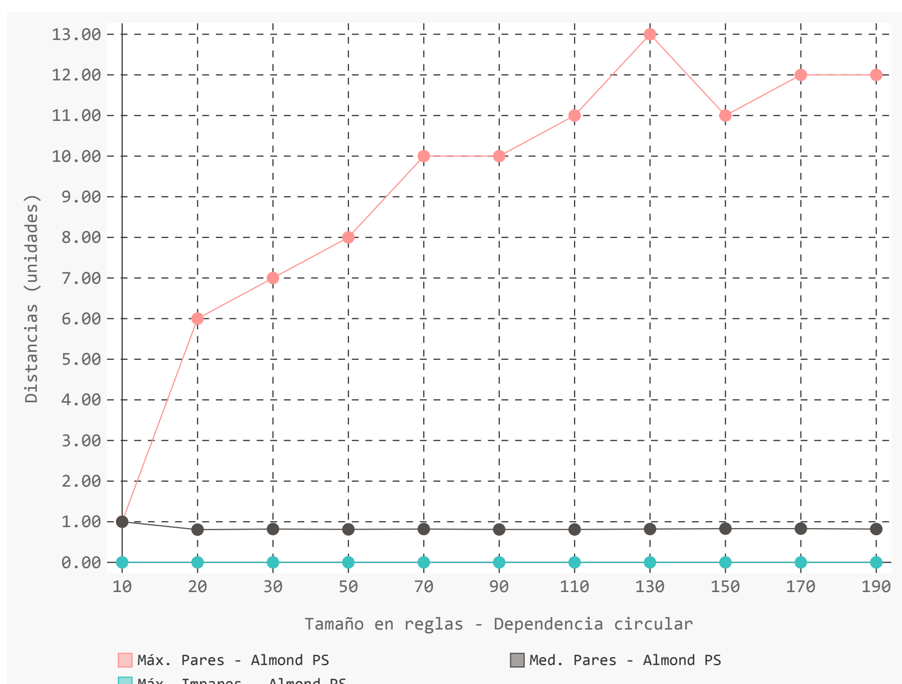


Fig. 5.9: Valor medio y máximo de la distancia de los resultados obtenidos en la ejecución, con la plataforma Almond PS, de los sistemas P de dependencia opuesta, respecto al valor más probable según la definición de la dependencia considerada. Para los objetos pares e impares se considera este valor como la unidad. Se emplean los sistemas de prueba con tamaños comprendidos entre 10 y 190 reglas.

tantes, presentan una distribución prácticamente exacta, con el primer cuartil en 0 unidades, mediana en 1 unidad y el tercero en 2 unidades. Respecto a los otros 4, las muestras se concentran en una multiplicidad 0 y 1, o en una multiplicidad 1 y 2, mostrando un comportamiento aproximado.

En la Fig. 5.9 se extienden, a todos los tamaños considerados de los sistemas P de prueba, los resultados obtenidos para los sistemas P de tamaño de 20 reglas. Se ha elegido un diagrama de líneas como medio de representación pero, a diferencia de los gráficos empleados en el análisis de las anteriores dependencias, no refleja las multiplicidades contextualizadas, sino los valores medios y máximos de las distancias de las multiplicidades de sus objetos. La razón es que esta dependencia, al igual que la circular, distribuye las multiplicidades de un modo semejante al uniforme, por lo que los valores contextualizados, para tamaños de reglas muy grandes,

tienden al valor nulo[2]. Se han considerado dos grupos de objetos, atendiendo a las características del modo de dependencia. El primer grupo recoge los objetos de índice par. Estos objetos son los compartidos entre dos parejas de reglas distintas, por lo que su comportamiento más probable es que fluctúe entre los valores de 0 y 1. El segundo grupo comprende los objetos de índice impar, los acumuladores de control. Estos acumuladores son compartidos por las mismas reglas de una pareja, que compiten por él, siendo su valor siempre 1, ya que es repuesto y consumido siempre por ambas reglas de la pareja. Así, y considerando que las distancias mostradas en el gráfico tienen como referencia, para ambos grupos de objetos, la unidad, se observa como las ejecuciones, en todos los tamaños considerados, se ajustan al comportamiento más probable. El grupo de los objetos de índice impar presenta distancias con valores máximos de 0, por lo que en todas las ejecuciones han tomado el valor 1. Atendiendo a los objetos de índice par, la distancia media es inferior a la unidad, lo que indica que presenta valores que fluctúan entre 0, 1 y 2. Los valores máximos de hasta 13 unidades reflejan un comportamiento normal, teniendo en cuenta la naturaleza no determinista de los sistemas P.

5.2.1.2. Comparación funcional de Almond PS y P-Lingua

Una vez realizado el análisis de las configuraciones finales reportadas por Almond PS, se contrastan estos resultados con los obtenidos con P-Lingua, considerada implementación de referencia. Esta sección se estructura del mismo modo que la anterior, analizando cada dependencia de forma independiente.

Dependencia circular

En primer lugar, la Fig. 5.10 muestra los resultados obtenidos con las dos implementaciones. Empleando los diagramas de cajas y bigotes utilizados anteriormente, se observa como ambas muestran valores similares, que se corresponden con los más probables, para el modo de dependencia considerado. Atendiendo a P-Lingua, la multiplicidad de la gran mayoría de los objetos fluctúa entre los valores 0, 1 y 2, a diferencia de los obtenidos con Almond PS, donde este tipo de variaciones, a

[2]Es preciso recordar que la suma de la multiplicidad de todos los objetos es constante para todas las transiciones, partiendo de una configuración inicial en la que cada objeto tiene una multiplicidad de 1. En este sentido, se puede decir que las multiplicidades no se crean ni se destruyen, sencillamente se distribuyen de un modo u otro por todos los objetos.

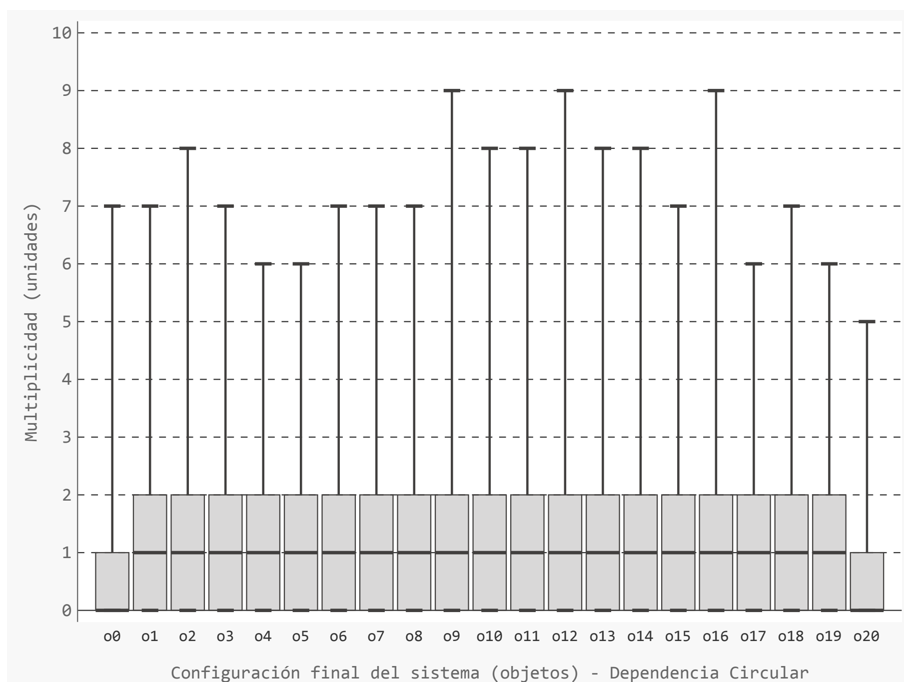
pesar de ser mayoría, es acompañado por un número significativo de objetos con multiplicidades que varían entre 0 y 1. Ambos resultados son igualmente válidos, considerando que el valor más probable es el valor 1.

En la Fig. 5.11 se muestran las distancias, respecto al valor más probable según la dependencia circular, de las multiplicidades de los objetos de los resultados conseguidos con ambas implementaciones, para sistemas de tamaño de 20 reglas. En este sentido, se observa como los resultados son similares, ofreciendo una dispersión en los valores máximos, justificados por la naturaleza no determinista de estos sistemas. No obstante, en este caso también se mantiene una tendencia similar en ambas implementaciones.

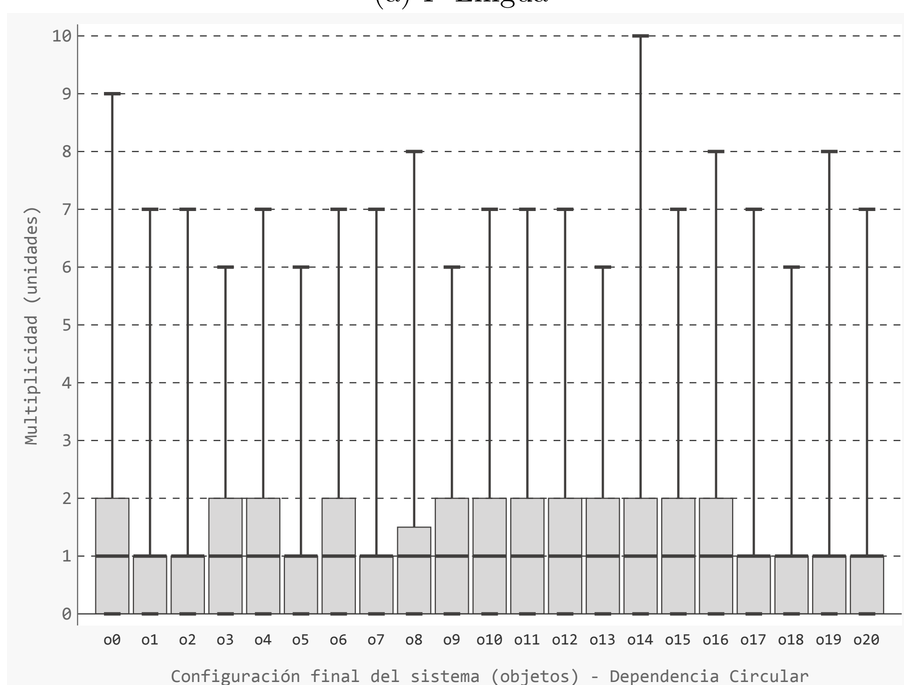
Como último resultado, la Fig. 5.12 extiende los resultados ofrecidos para el tamaño de 20 reglas a todos los tamaños consideradores en el conjunto de sistemas de prueba. En este sentido, es preciso indicar que se muestran los valores medios y máximos de las multiplicidades de los objetos, y no de las absolutas. Tal y como sucede para el tamaño de 20, los resultados de ambas implementaciones son similares, no existiendo diferencias significativas. Así, la tendencia es muy similar para los resultados máximos, mientras que para los medios, aunque también lo es, Almond PS presenta, mayoritariamente, las mayores distancias medias, mientras que, a su vez, presenta el mejor comportamiento medio. Estos resultados confirman la similitud de los resultados entre las dos implementaciones, ya que si bien la distancia ofrecida por Almond PS respecto al valor más probable es inferior, la diferencia no es lo suficientemente significativa como para ser relevante.

Dependencia circular-2

A continuación se analiza la dependencia circular-2. En primer lugar, se muestran en la Fig. 5.13 los resultados obtenidos para la ejecución, con la implementación *software* P-Lingua, de un sistema (del conjunto de prueba) con un tamaño de 20 reglas. Con el propósito de facilitar la comparación entre ambas implementaciones, se muestra conjuntamente con los obtenidos con la arquitectura Almond PS, ya mostrados en la sección anterior. En la gráfica se observa como ambas implementaciones presentan un comportamiento correcto, ajustándose a los resultados



(a) P-Lingua



(b) Almond PS

Fig. 5.10: Diagrama de cajas y bigotes que muestra los cuartiles $Q0$ (valor mínimo), $Q1$, $Q2$ (mediana), $Q3$ y $Q4$ (valor máximo) de la distribución seguida por las multiplicidades de los objetos tras la ejecución de los sistemas P con dependencia circular. En (a) se representan los resultados obtenidos con P-Lingua y en (b) los obtenidos con Almond PS, para facilitar la comparación entre ambos.

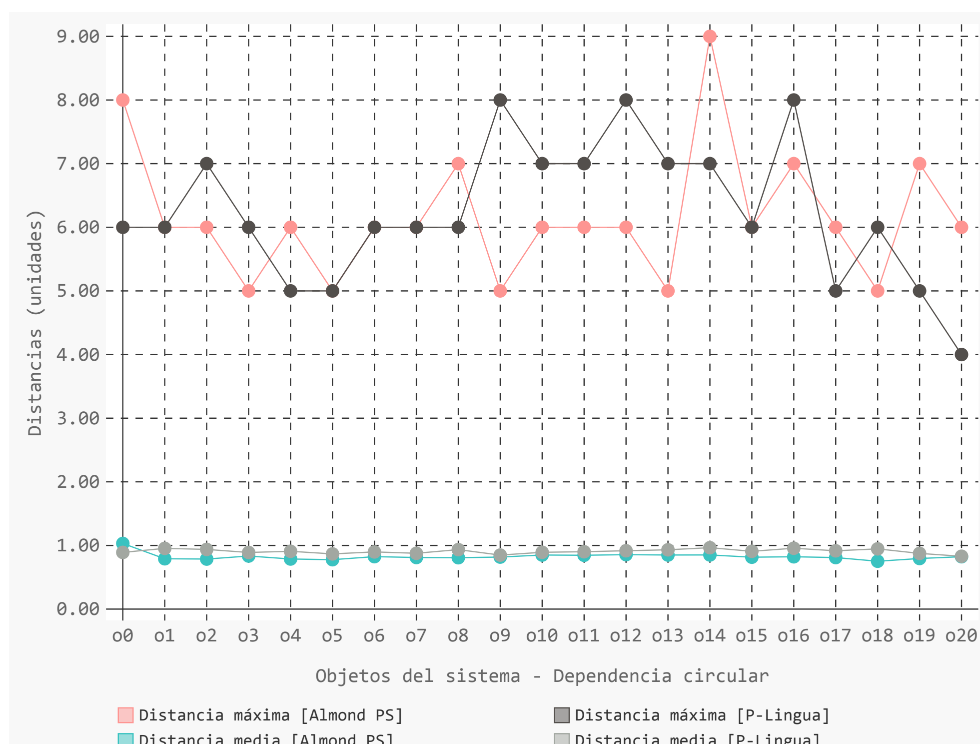


Fig. 5.11: Valor medio y máximo de las medias de las distancias de sistemas P, con dependencia circular y tamaño de 20 reglas. La figura muestra las distancias correspondientes a P-Lingua y Almond PS. Este último resultado con el propósito de facilitar la comparación entre ambas implementaciones.

más probables según las características del modo de dependencia. No se aprecia, en consecuencia, ninguna diferencia relevante entre Almond PS y P-Lingua.

Los resultados anteriores se extienden para los tamaños comprendidos entre 10 y 190 reglas en la Fig. 5.14. Para ello, se muestra la multiplicidad media contextualizada de las medias de las multiplicidades de los objetos de los sistemas P. Al igual que en la figura anterior, se muestran los resultados de ambas implementaciones para facilitar su contraste. En la dependencia circular-2, los objetos extremos, o_0 y o_N, actúan de acumuladores de la multiplicidad del sistema, siendo el primero el que concentra la mayor multiplicidad. Sus adyacentes, o_1 y o_{N-1}, se agotan en todas las ejecuciones y, respecto al resto de objetos, tienden a presentar una multiplicidad nula, aunque el carácter no determinista de estos sistemas, generalmente motiva que algunos de ellos conserven algunas multiplicidades. Teniendo

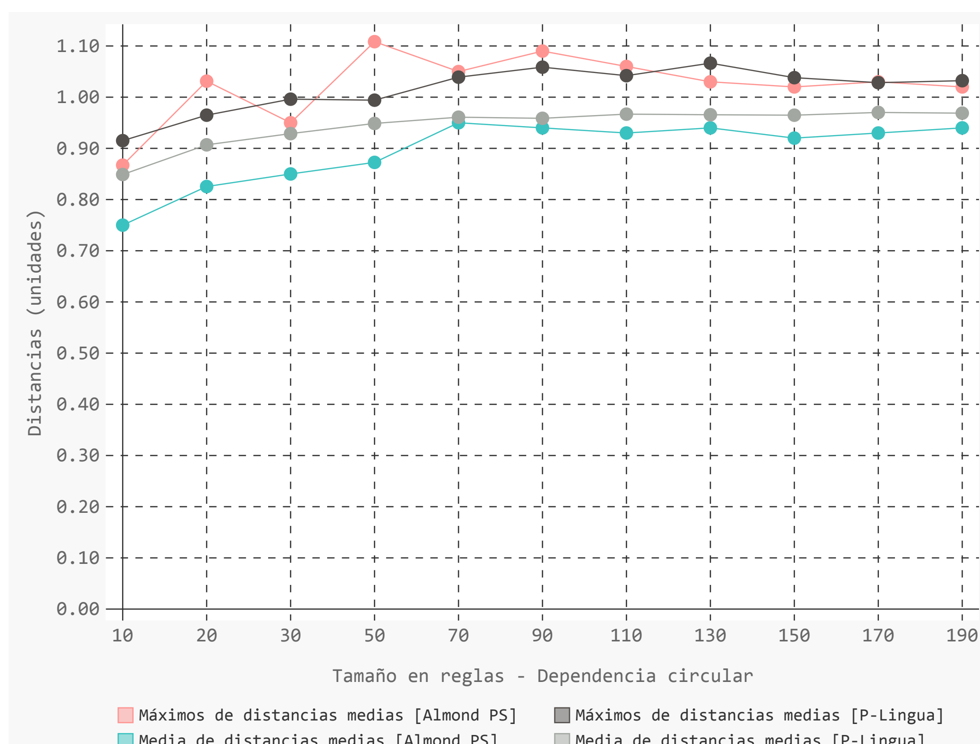
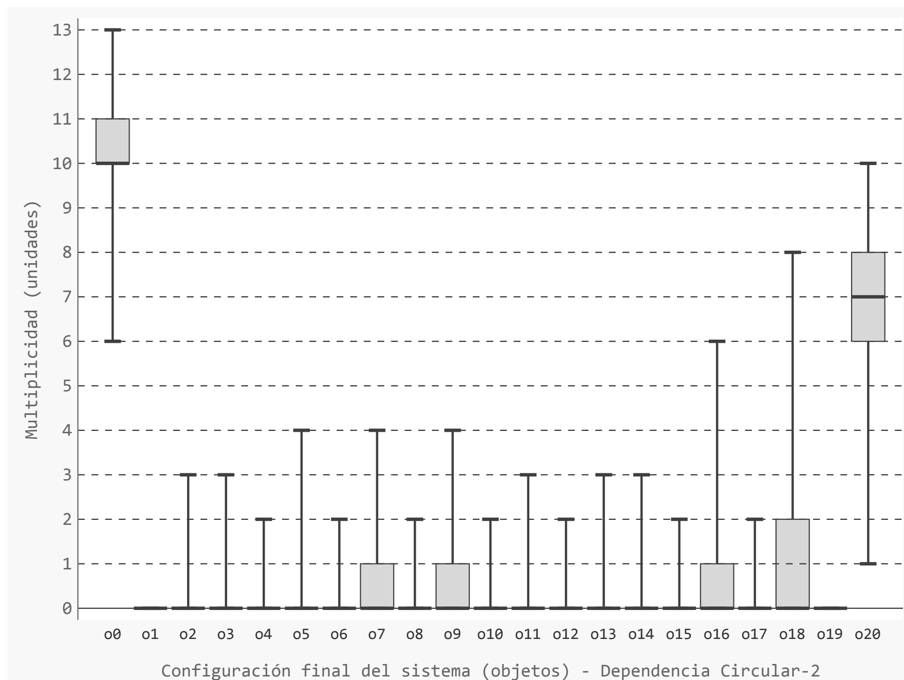


Fig. 5.12: Valor medio y máximo de las medias de las distancias de las multiplicidades de los objetos tras la ejecución de los sistemas P de prueba, con dependencia circular y tamaño de 10 a 190 reglas. La figura muestra las distancias correspondientes a P-Lingua y Almond PS. Este último resultado con el propósito de facilitar la comparación entre ambas implementaciones.

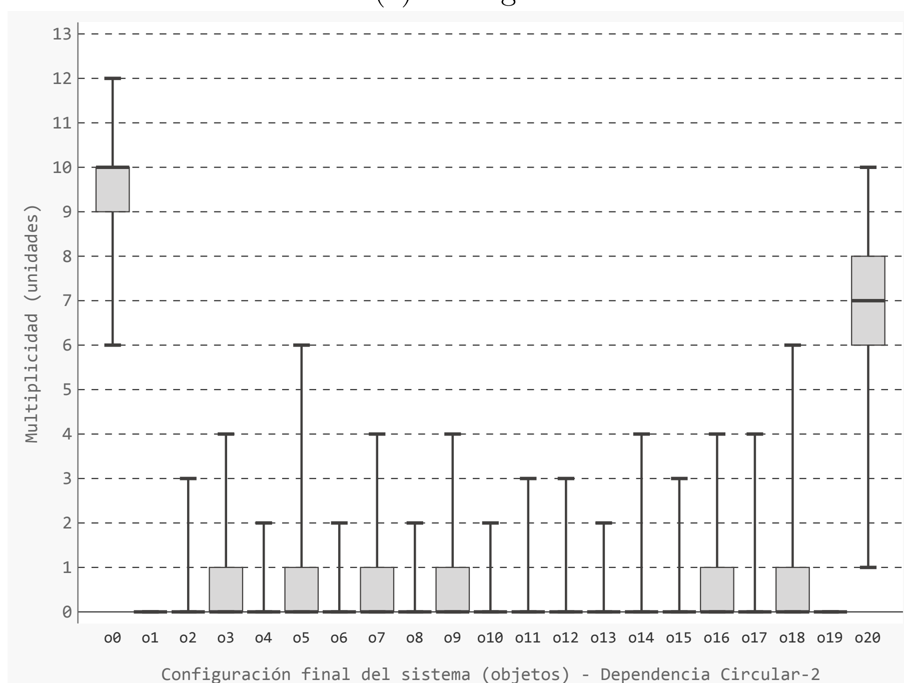
en cuenta estas características, los objetos se distribuyen en cuatro grupos en la representación gráfica de los resultados. En esta representación se observan resultados similares para ambas implementaciones, ajustándose al comportamiento más probable. No se aprecian diferencias significativas entre ambas implementaciones, si bien en P-Lingua se observa una concentración ligeramente superior en los objetos acumuladores, mientras que la implementación Almond PS presenta una distribución de las multiplicidades ligeramente más repartida por el resto de objetos.

Dependencia lineal

La dependencia lineal es la próxima en ser analizada. En primer lugar, se muestran en la Fig. 5.15 los resultados obtenidos tras la ejecución, con P-Lingua, de



(a) P-Lingua



(b) Almond PS

Fig. 5.13: Diagrama de cajas y bigotes que muestra los cuartiles $Q0$ (valor mínimo), $Q1$, $Q2$ (mediana), $Q3$ y $Q4$ (valor máximo) de la distribución seguida por las multiplicidades de los objetos tras la ejecución de los sistemas P con dependencia circular-2. En (a) se representan los resultados obtenidos con P-Lingua y en (b) los obtenidos con Almond PS, para facilitar la comparación entre ambos.

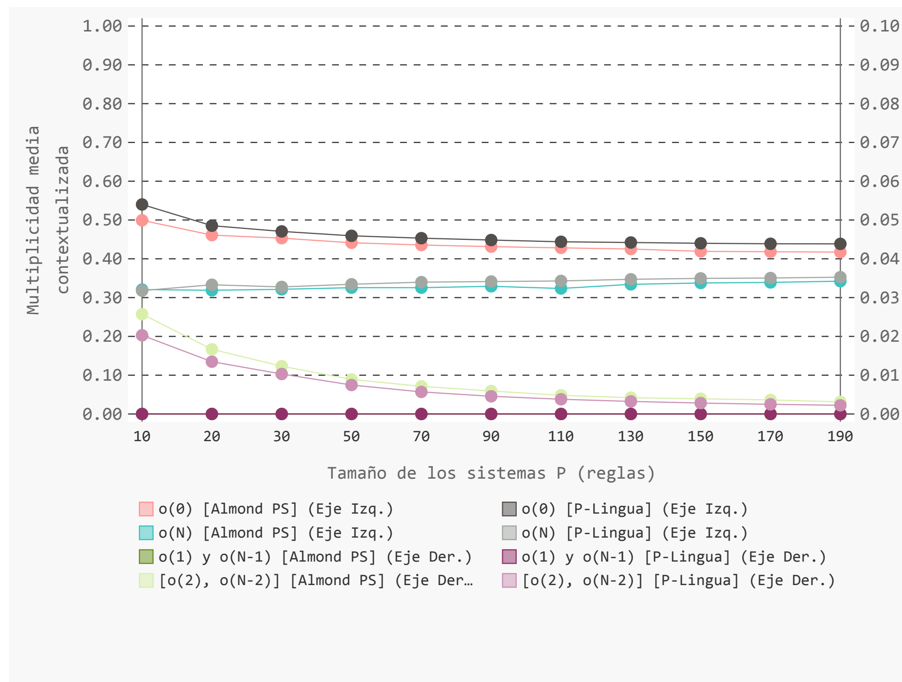


Fig. 5.14: Multiplicidad media contextualizada (multiplicidad media dividida entre la multiplicidad total presente en el sistema) de las medias de las multiplicidades de los objetos de los sistemas P, con tamaños comprendidos entre 10 y 190 reglas y dependencia circular-2, tras su ejecución bajo las implementaciones Almond PS y P-Lingua. Para cada tamaño se muestra la multiplicidad para los objetos más característicos: los extremos (o_0 y o_N) y sus adyacentes (o_1 y o_{N-1}), así como el resto de objetos agrupados. La representación de los valores para los objetos o_1 y o_{N-1} de ambas implementaciones se superponen.

un sistema P con un tamaño de 20 reglas. También se adjuntan los resultados para Almond PS ya presentados, con el propósito de facilitar la comparación entre ambos sistemas. En esta dependencia, las multiplicidades se concentran en el último objeto, o_N, que actúa de acumulador, sin existir realimentación del propio sistema. Tal y como sucede en la dependencia anterior, el objeto adyacente al acumulador, o_{N-1}, se agota en todas las ejecuciones. En relación al resto de objetos, sus multiplicidades más probables se corresponden con el valor nulo, aunque el no determinismo inherente a los sistemas P hace que se distribuyan algunas pocas multiplicidades entre ellos. En la Fig. 5.15 se observa como ambas implementaciones presentan resultados muy similares, ajustándose al comportamiento

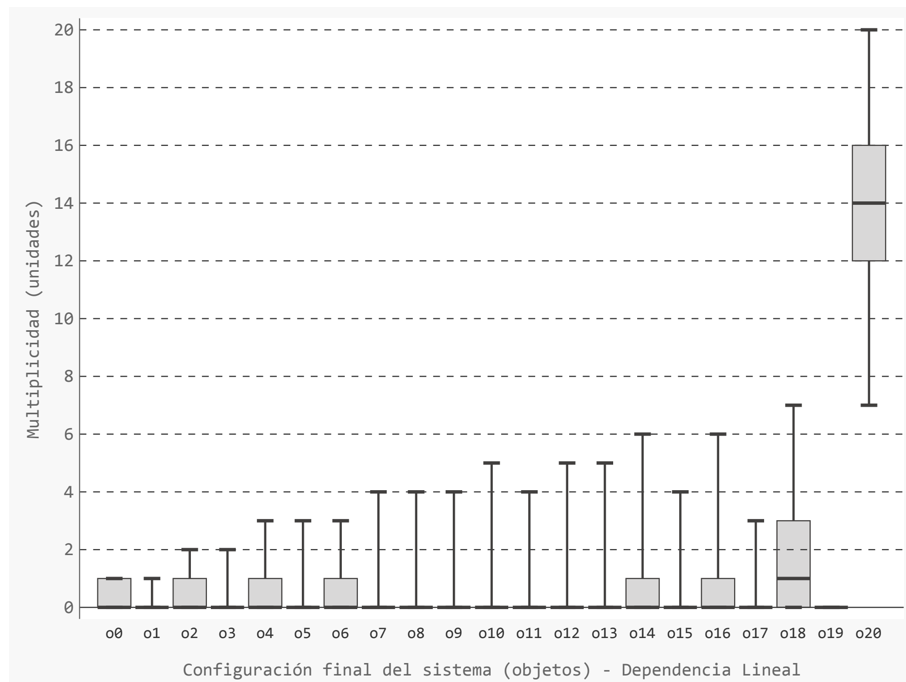
característico de esta dependencia. No se observan, por lo tanto, diferencias dignas de mención entre ambas plataformas.

Como en los casos anteriores, en la Fig. 5.16 se observa que el comportamiento mostrado para el tamaño de 20 reglas se repite para todos los tamaños considerados en el conjunto de sistemas de prueba. Con ese propósito muestra la multiplicidad media contextualizada de las medias de las multiplicidades de los objetos. Junto con los resultados de P-Lingua, se muestran los ya analizados previamente de Almond PS, para facilitar su comparación. Los comportamientos mostrados son similares, y conforme a los valores más probables. Los grupos considerados son cuatro, comprendiendo, en este orden, el primer objeto, el último, el penúltimo y el resto de objetos. No se aprecian diferencias significativas, aunque, como ocurre en otro tipo de dependencias, Almond PS presenta una multiplicidad ligeramente inferior en el acumulador, siendo la distribución de multiplicidades entre el resto de los objetos levemente superior.

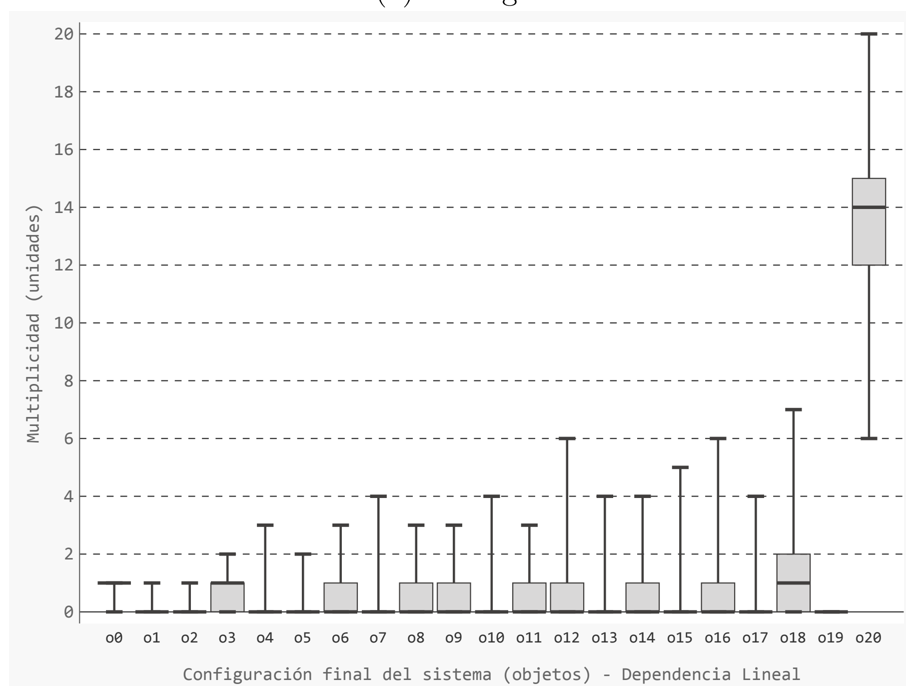
Dependencia opuesta

En último lugar se analiza la dependencia opuesta. Como en los casos anteriores, se muestran los resultados en la Fig. 5.17, tras la ejecución, con P-Lingua, de un sistema P con un tamaño de 20 reglas. Se representan también los obtenidos con Almond PS, para facilitar la comparación entre ambas implementaciones. En esta dependencia, los objetos de índice impar actúan de elementos de control, presentando siempre una multiplicidad igual a la unidad, mientras que los objetos de índice par fluctúan entre los valores más probables de 0, 1 y 2. En este sentido, la representación gráfica de ambas implementaciones presenta un comportamiento que se corresponde con el más probable para este tipo de dependencia. Sin embargo, se observa como con la implementación Almond PS, los valores de multiplicidad de los objetos se distribuyen de una forma más equitativa entre los valores más probables, a diferencia de P-Lingua, que tiende a mostrar multiplicidades con un valor de 0 ó 1 para este grupo de objetos. No obstante, no se considera una diferencia relevante entre ambas plataformas.

La Fig. 5.18 extiende los resultados anteriormente presentados para el resto de tamaños del conjunto de sistemas de prueba considerado. En ella, se muestra el valor medio y máximo de la distancia de los resultados obtenidos, tomando como



(a) P-Lingua



(b) Almond PS

Fig. 5.15: Diagrama de cajas y bigotes que muestra los cuartiles $Q0$ (valor mínimo), $Q1$, $Q2$ (mediana), $Q3$ y $Q4$ (valor máximo) de la distribución seguida por las multiplicidades de los objetos tras la ejecución de los sistemas P con dependencia lineal. En (a) se representan los resultados obtenidos con P-Lingua y en (b) los obtenidos con Almond PS, para facilitar la comparación entre ambos.

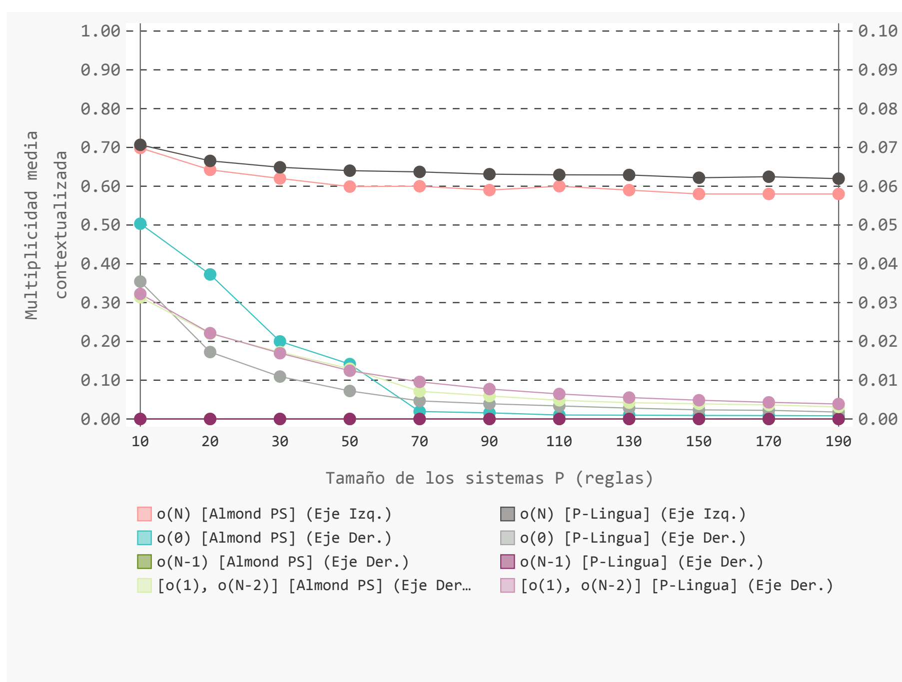
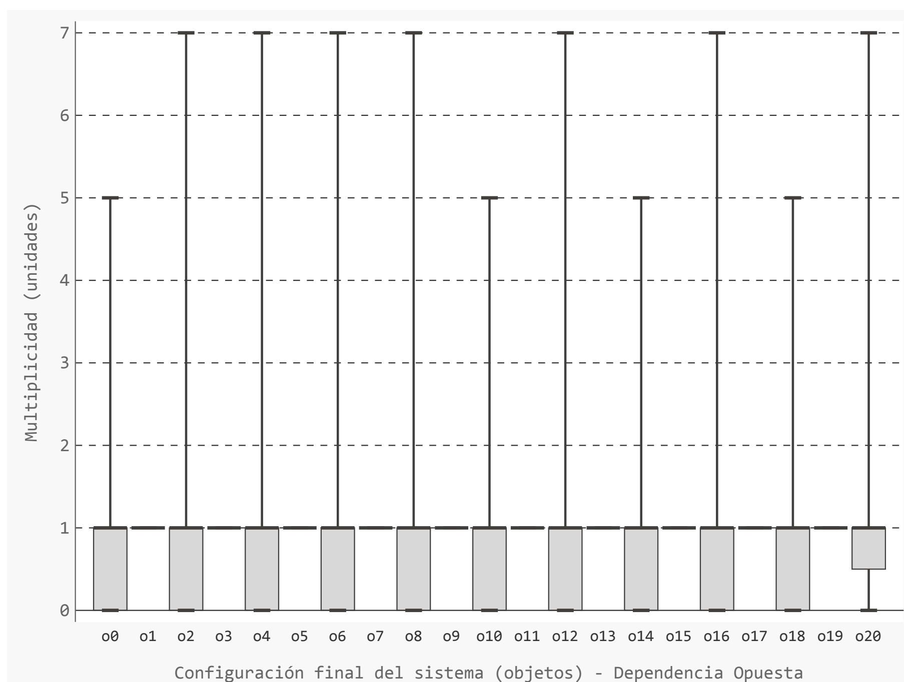
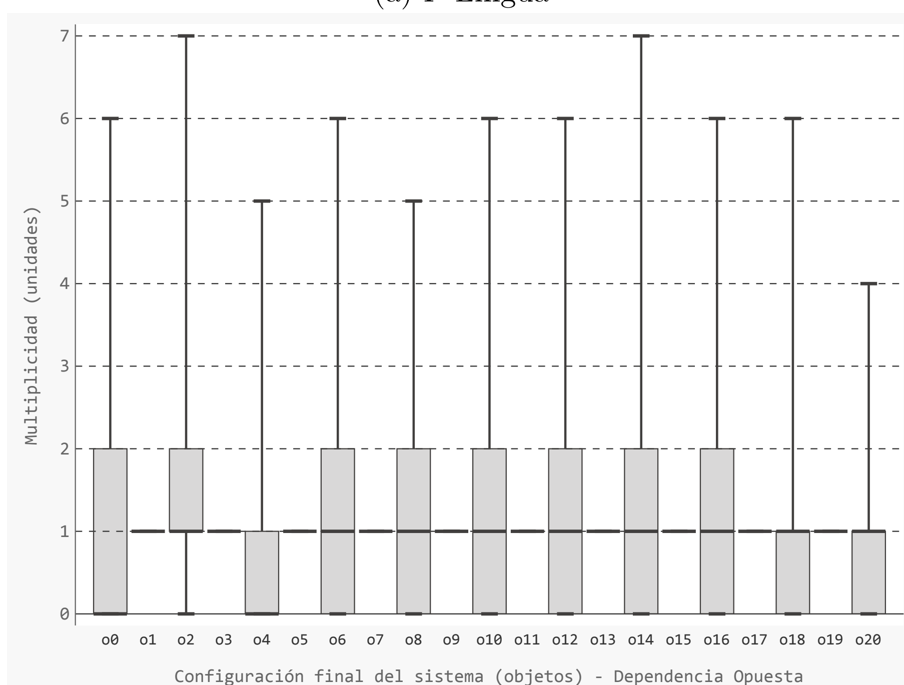


Fig. 5.16: Multiplicidad media contextualizada (multiplicidad media dividida entre la multiplicidad total presente en el sistema) de las medias de las multiplicidades de los objetos de los sistemas P, con tamaños comprendidos entre 10 y 190 reglas y dependencia lineal, tras su ejecución bajo las implementaciones P-Lingua y Almond PS. Para cada tamaño se muestra la multiplicidad para los objetos más característicos: los extremos (o_0 y o_N) y o_{N-1}, así como el resto de objetos agrupados. La representación de los valores para el objeto $o_{(N-1)}$ de ambas implementaciones se superponen, ya que se obtiene el mismo valor en ambas.

valor de referencia, o valor más probable, la unidad. Los objetos son agrupados según su paridad. Así, se observa como ambas implementaciones presentan resultados válidos y muy similares, tomando los objetos de índice impar una multiplicidad constante igual a 1, representada como una distancia nula. Respecto a los objetos de índice par, Almond PS presenta una distancia media ligeramente superior, siguiendo la misma línea que los resultados mostrados anteriormente, en el sentido de una distribución más equitativa de los valores de las multiplicidades, en torno a los más probables, una veces ligeramente inferior, otras superior, aunque sin mostrar una diferencia relevante. Por último, los valores máximos siguen una tendencia similar para ambas plataformas.



(a) P-Lingua



(b) Almond PS

Fig. 5.17: Diagrama de cajas y bigotes que muestra los cuartiles $Q0$ (valor mínimo), $Q1$, $Q2$ (mediana), $Q3$ y $Q4$ (valor máximo) de la distribución seguida por las multiplicidades de los objetos tras la ejecución de los sistemas P con dependencia opuesta. En (a) se representan los resultados obtenidos con P-Lingua y en (b) los obtenidos con Almond PS, para facilitar la comparación entre ambos.

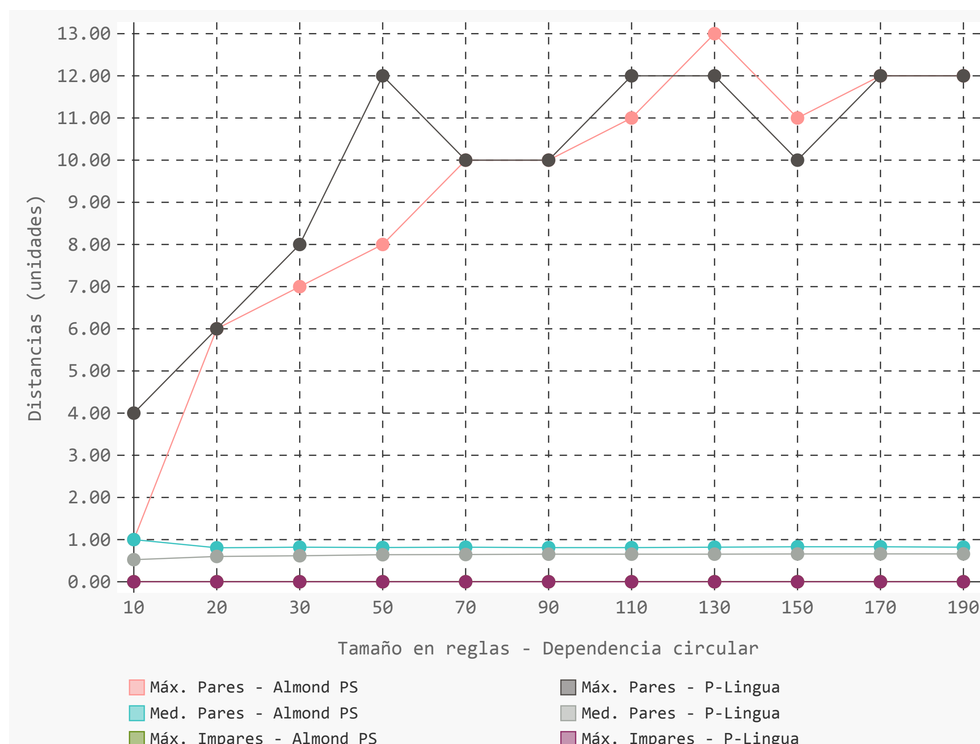


Fig. 5.18: Valor medio y máximo de la distancia de los resultados obtenidos en la ejecución, con las implementaciones P-Lingua y Almond PS, de los sistemas P de dependencia opuesta, respecto al valor más probable según la definición de la dependencia considerada. Para los objetos pares e impares se considera este valor como la unidad. La ejecución considera los sistemas de prueba, con tamaños comprendidos de 10 a 190 reglas. La representación de los resultados de los valores máximos de los objetos de índice impar de ambas implementaciones se superpone, ya que representan el mismo valor.

5.2.2. Análisis de las transiciones necesarias para alcanzar la condición de parada total

Una vez analizadas las multiplicidades finales que muestran los sistemas P, es preciso observar el número de transiciones necesarias para alcanzar una condición de parada total. Para este análisis es preciso recordar los modos de dependencia contemplados en los sistemas de prueba. De los cuatro presentados, dos de ellos, el de dependencia circular-2 y lineal, deben alcanzar una condición de parada total, mientras que los sistemas con dependencia circular y opuesta nunca deben

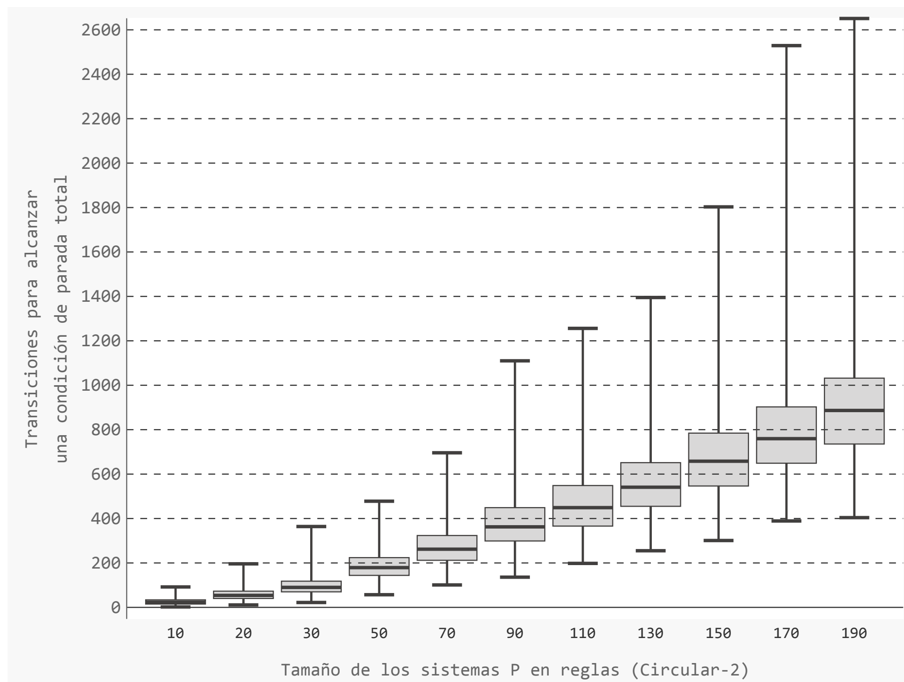
alcanzar esta condición, al no converger hacia una configuración final. Al igual que en la sección anterior, los resultados adquiridos de la arquitectura Almond PS y P-Lingua se comparan conjuntamente.

Respecto a los modos de dependencia circular y opuesta, los resultados experimentales confirman la no convergencia de estos sistemas. En todos los casos, los sistemas P alcanzan una condición de parada límite de transiciones. Esta condición tiene lugar cuando se ejecutan 8192 transiciones sin que el sistema converja hacia una configuración de parada, según el escenario de prueba contemplado. Cuando los sistemas son ejecutados con P-Lingua se obtienen los mismos resultados. Estos resultados no se muestran gráficamente, por no considerarse su representación lo suficientemente significativa.

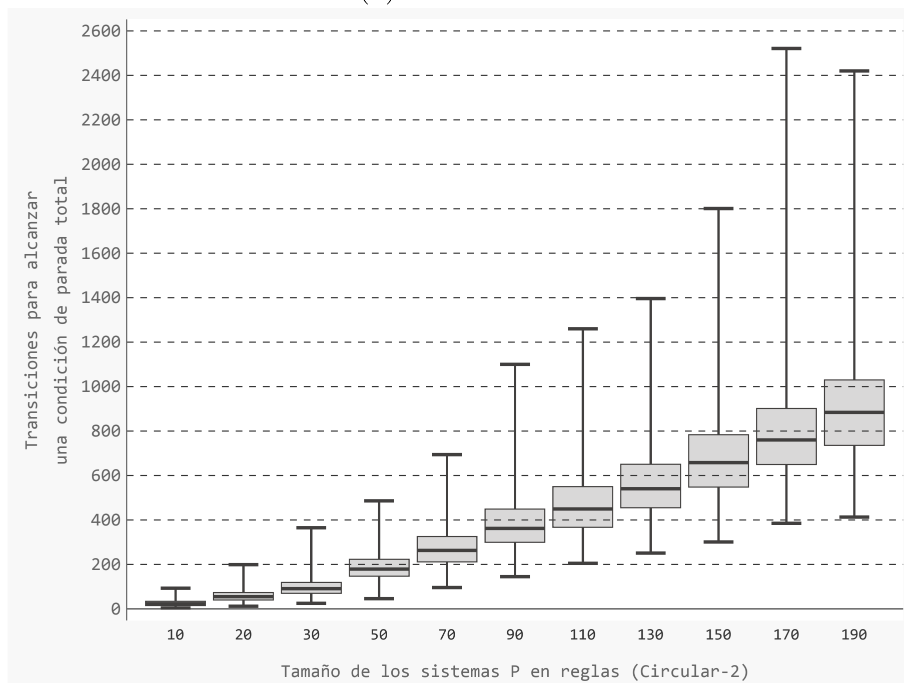
Respecto a los sistemas con dependencia circular-2, la Fig. 5.19 muestra como alcanzan una condición de parada total en todas las ejecuciones. Este comportamiento coincide con el mostrado por P-Lingua, si bien Almond PS muestra una mayor dispersión en el número de transiciones necesarias. Este hecho puede ser un indicio de presentar un comportamiento no determinista más marcado. No obstante, la diferencia no es significativa.

Los resultados para el modo de dependencia lineal son similares a los obtenidos para el modo circular-2, arrojando ambas implementaciones los resultados esperados, esto es, una condición de parada total en todas las ejecuciones, tal y como se muestra en la Fig. 5.20 . Al igual que en el caso anterior, Almond PS muestra una dispersión ligeramente mayor.

Entre ambos modos de dependencia, la diferencia más significativa es la velocidad de convergencia, siendo los sistemas de dependencia lineal los más rápidos. Esta diferencia en el comportamiento de ambos sistemas está motivada por su definición. En este sentido, es preciso recordar que, para la dependencia lineal, la relación generados/consumidos es de 0/1 y 1/2 para los objetos de menor índice, mientras que para el acumulador, el objeto de mayor índice, de 3/1, siendo para los restantes 2/2, por lo que se trata de la relación que mueve con mayor rapidez los objetos en dirección al acumulador. Por su parte, los objetos de dependencia circular-2 presentan una relación generados/consumidos de 2/1 para el objeto de menor índice, o_0, 1/2 para el segundo objeto de menor índice, o_1, 2/1 para el objeto de mayor índice o_N, y 2/2 para el resto de los objetos.

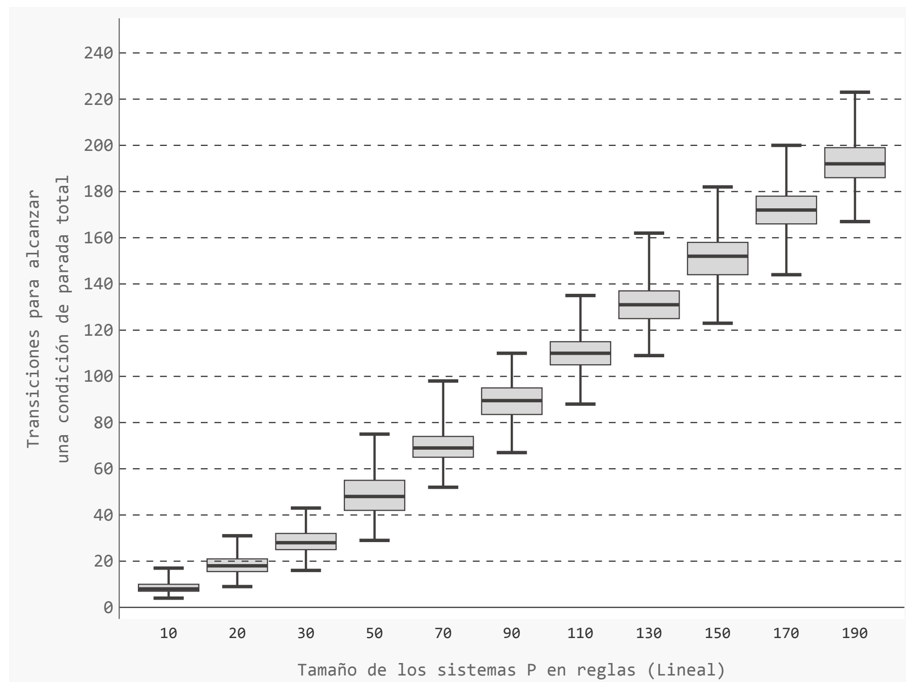


(a) Almond PS

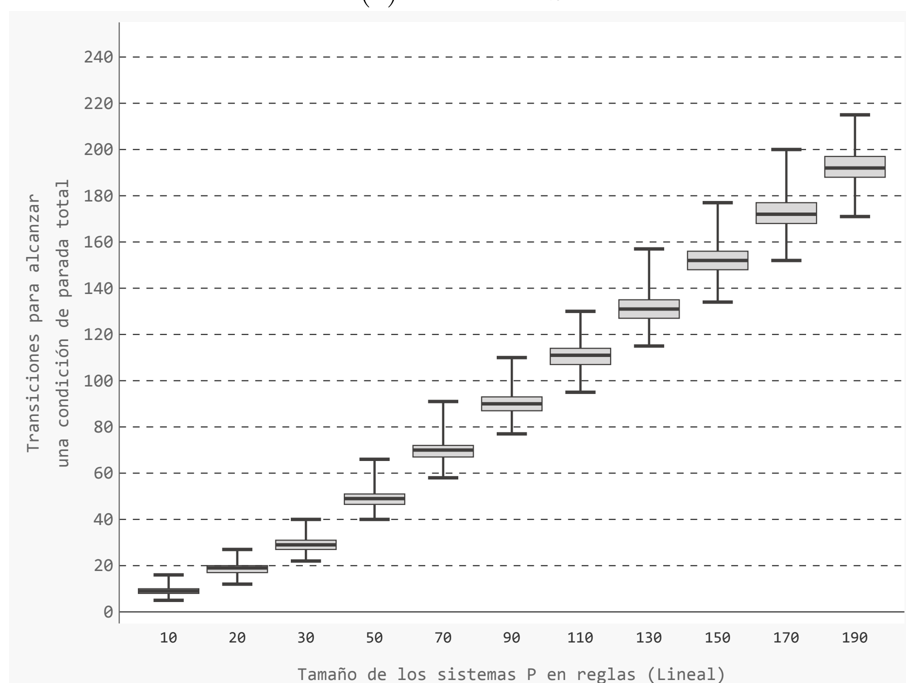


(b) P-Lingua

Fig. 5.19: Número de transiciones para que el sistema P alcance una condición de parada total. Los sistemas P empleados siguen una dependencia circular-2, y han sido ejecutados con la implementación Almond PS (a) y P-Lingua (b). Representación de diagrama de cajas y bigotes que muestra los cuartiles $Q0$ (valor mínimo), $Q1$, $Q2$ (mediana), $Q3$ y $Q4$ (valor máximo) de la distribución seguida por el número de transiciones.



(a) Almond PS



(b) P-Lingua

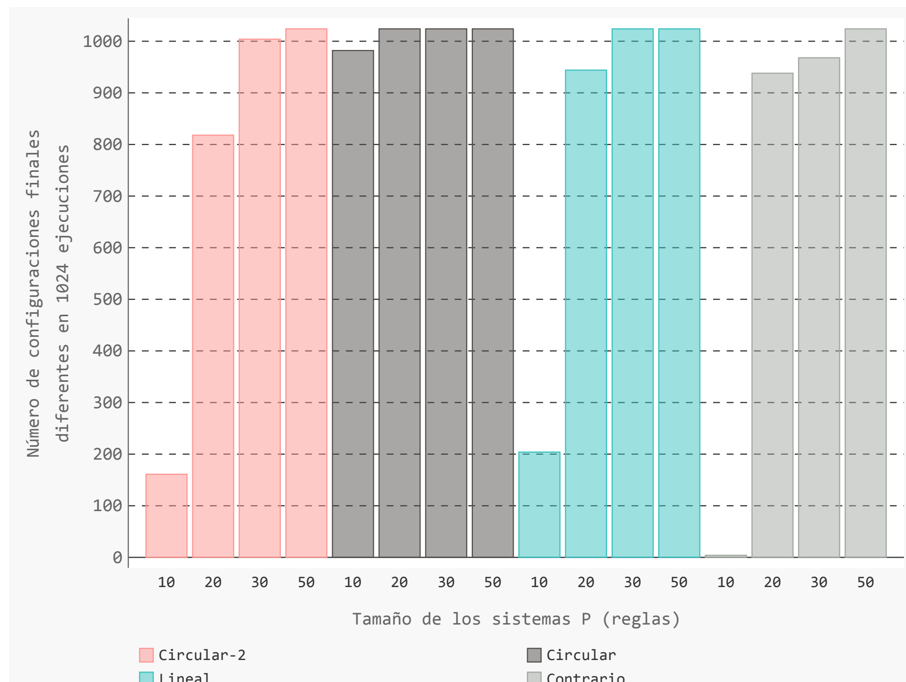
Fig. 5.20: Número de transiciones para que el sistema P alcance una condición de parada total. Los sistemas P empleados siguen una dependencia lineal, y han sido ejecutados con la implementación Almond PS (a) y P-Lingua (b). Representación de diagrama de cajas y bigotes que muestra los cuartiles $Q0$ (valor mínimo), $Q1$, $Q2$ (mediana), $Q3$ y $Q4$ (valor máximo) de la distribución seguida por el número de transiciones.

Otra diferencia apreciable es la diferencia en el número de resultados diferentes, siendo los resultados de la dependencia circular-2 los que presentan una mayor dispersión. La razón es la misma que en el caso anterior: la definición de los sistemas. Atendiendo a la dependencia circular-2, existen dos puntos críticos que contribuyen a la parada del sistema. Tomando como ejemplo el sistema mostrado en la Fig. 5.4, los dos puntos son el objeto o_1, que rompe la dependencia circular, y el o_{19}, cuyo agotamiento, en segundo lugar, supone la parada del sistema. Este hecho hace que la dependencia circular-2 muestre una dispersión ligeramente mayor en los resultados, especialmente en la diferencia de los valores máximos y mínimos mostrados.

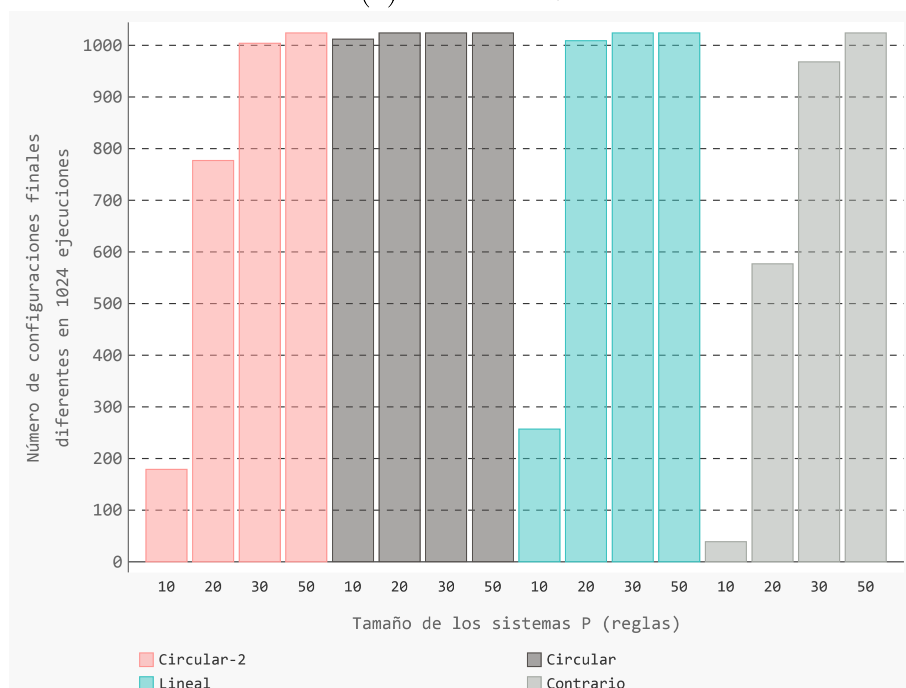
5.2.3. Análisis de la distribución en el espacio de solución de las configuraciones finales

El último aspecto a analizar está relacionado con la distribución seguida por las configuraciones finales alcanzadas. En esta sección, al igual que en la anterior, se presentan conjuntamente el análisis de los resultados experimentales de Almond PS, y su comparación con los obtenidos con P-Lingua.

La Fig. 5.21 muestra, para ambas implementaciones, el número de configuraciones finales diferentes para cada uno de los modos de dependencia, considerando los tamaños 10, 20, 30 y 50. Para tamaños mayores, no existe ningún elemento repetido en los resultados adquiridos, aunque dado el carácter no determinista de los sistemas P, en otra batería de adquisición de resultados, podrían tener lugar algunas repeticiones, aunque sin entidad suficiente. Antes de continuar, es preciso recordar que para la obtención de los resultados, los sistemas P empleados utilizan una distribución equiprobable para la generación del conjunto *Appl*, así como en la selección del conjunto R, Sección 3.3 (Pág. 95). Es por ello que los resultados, relativos a las 1024 ejecuciones diferentes, deben estar distribuidos uniformemente entre el espacio de soluciones, sujeto a las restricciones impuestas por el modo de dependencia. Así, la dependencia circular es la que tiene una probabilidad de no repetición superior, ya que distribuye las multiplicidades de forma equitativa entre los distintos objetos, obteniendo configuraciones distintas para sistemas con un tamaño igual o superior a 20 reglas.



(a) Almond PS



(b) P-Lingua

Fig. 5.21: Número de configuraciones finales diferentes para tamaños de 10, 20, 30 y 50 reglas, de cada uno de los sistemas P de referencia. Se muestran los resultados obtenidos con Almond PS (a) y P-Lingua (b).

En relación a la dependencia opuesta, presenta un bajo índice de dispersión, aún considerando su no convergencia, para el tamaño de 10 reglas. Esto es debido a la existencia de gran cantidad de objetos de control, en este caso, aquellos de índice impar que mantienen constante su multiplicidad a 1. De ese modo, estos objetos se mantienen fijos, mientras que los objetos pares van variando de una transición a otra, reduciendo el número de configuraciones finales diferentes a ser alcanzadas. El resultado cambia completamente para un tamaño de 20 reglas, mejorando para 30 y alcanzando una dispersión total para tamaños iguales o superiores a 50.

Por último, las dependencias circular-2 y lineal, obtienen un bajo índice de dispersión para tamaños pequeños, debido a su convergencia y al establecimiento de objetos de control que, en este caso, hacen referencia al agotamiento de sus multiplicidades. En este sentido, es claramente perceptible como la existencia de 2 objetos de control en la dependencia circular-2, frente a 1 de la dependencia lineal, provoca que haya más repeticiones en el primer caso, esto es, para aquellos sistemas de dependencia circular-2.

Comparando estos resultados con los obtenidos con P-Lingua, para tamaño de sistemas de 30 reglas los resultados son prácticamente iguales, y a partir de tamaños de 50, el espacio de soluciones es lo suficientemente grande como para que no existan repeticiones significativas. Para tamaños de 10 y 20 reglas, P-Lingua ofrece, ligeramente, una mayor dispersión de los resultados, al existir un mayor número de configuraciones finales no repetidas, en las dependencias lineal, y circular. En contraposición, en la dependencia opuesta y circular-2 es Almond PS la solución que presenta una mayor dispersión. Estos resultados están motivados por el uso de generadores de números aleatorios distintos.

Estos resultados confirman el correcto funcionamiento del sistema para los casos analizados, según el comportamiento esperado por los modos de dependencia empleados. Así se obtiene que para tamaños superiores o iguales a 50 reglas, las soluciones se distribuyen uniformemente entre el espacio de posibles soluciones, verificando unas de las principales características de los sistemas P. Al mismo tiempo, este comportamiento es ratificado por P-Lingua.

5.3. Recursos *hardware*

Una vez estudiados los resultados funcionales de la arquitectura Almond PS, es preciso analizar los recursos *hardware* empleados por la implementación presentada. Con el propósito de mostrar unos resultados lo más exhaustivos posibles, el estudio se estructura en base a cinco grados de libertad, según las características de los sistemas P aceptados, la arquitectura Almond PS y la tecnología de implementación.

Atendiendo a los recursos *hardware* analizados, se consideran los recursos de tipo LUT, *Slice*, *Block RAM* (BRAM) y *Digital Signal Processor* (DSP). Se remite a la Sección 1.1.3.1 (Pág. 22) si se desea obtener más información.

A continuación se describen brevemente estos cincos parámetros de variación:

- Relacionados con los sistemas P de entrada.

  Dependencias entre reglas Además del número de reglas del sistema, es preciso analizar las dependencias que existen entre las distintas reglas. Con este propósito, se han tomado como elementos de prueba las dependencias entre reglas descritas en la Sección 5.1 (Pág. 155): Sistema Circular, Sistema Circular-2, Sistema Opuesto y Sistema Lineal.

  Complejidad del sistema de entrada Este parámetro hace referencia directa al número de entidades presentes en el sistema: número de reglas y objetos. Tomando como referencia los sistemas P de entrada descritos en la Sección 3.3 (Pág. 95), el número de objetos se encuentra determinado por el número de reglas del sistema. Es por ello que se tomará como medida de la complejidad del sistema P su número de reglas.

- Relacionados con la arquitectura Almond PS.

  Resolución de cardinalidades La máxima cardinalidad representable es una restricción impuesta por la implementación Almond PS. Esta restricción viene determinada por el tamaño de los buses empleados en el diseño *hardware*, que incidirá en los recursos empleados en la ruta de datos de la arquitectura.

- Relacionados con la tecnología de implementación.

Modo de generación software El código HDL producido por el *software* de generación es interpretado por las herramientas del fabricante de la FPGA, encargadas de obtener el fichero de configuración de la FPGA o *bitstream*. Aunque su funcionamiento interno es ajeno a terceros, se ha considerado interesante el estudio de como afectan las distintas opciones de optimización al uso de los recursos de la FPGA.

Tecnologías de FPGA La tecnología FPGA se encuentra en constante evolución, mejorando su comportamiento en cada generación. En este sentido, se ha considerado de interés la comparación de distintas tecnologías de FPGA.

Por último, los sistemas P de entrada empleados son los descritos en la Sección 5.1 (Pág. 155).

5.3.1. Recursos *hardware* según dependencias entre reglas

En primer lugar, se analiza el impacto en los recursos *hardware* al modificar las relaciones de dependencia entre las reglas. Estos resultados son mostrados en la Fig. 5.22, empleando un gráfico de líneas en el que, considerando los tamaños de 10 a 190 reglas, se recogen los valores para los cuatro modos de dependencia considerados. En este sentido, se observa como la implementación de las distintas dependencias no afecta al consumo de ninguno de los recursos de la FPGA. Se trata de un resultado esperado, ya que distintos modos de generación conservan la estructura general del conjunto de reglas, así como su parte izquierda, empleando magnitudes similares en los recursos generados por su parte derecha.

En consecuencia, al no influir el modo de dependencias entre reglas en el consumo de recursos, para el resto de las secciones de este apartado únicamente se contemplan los sistemas P de dependencia circular.

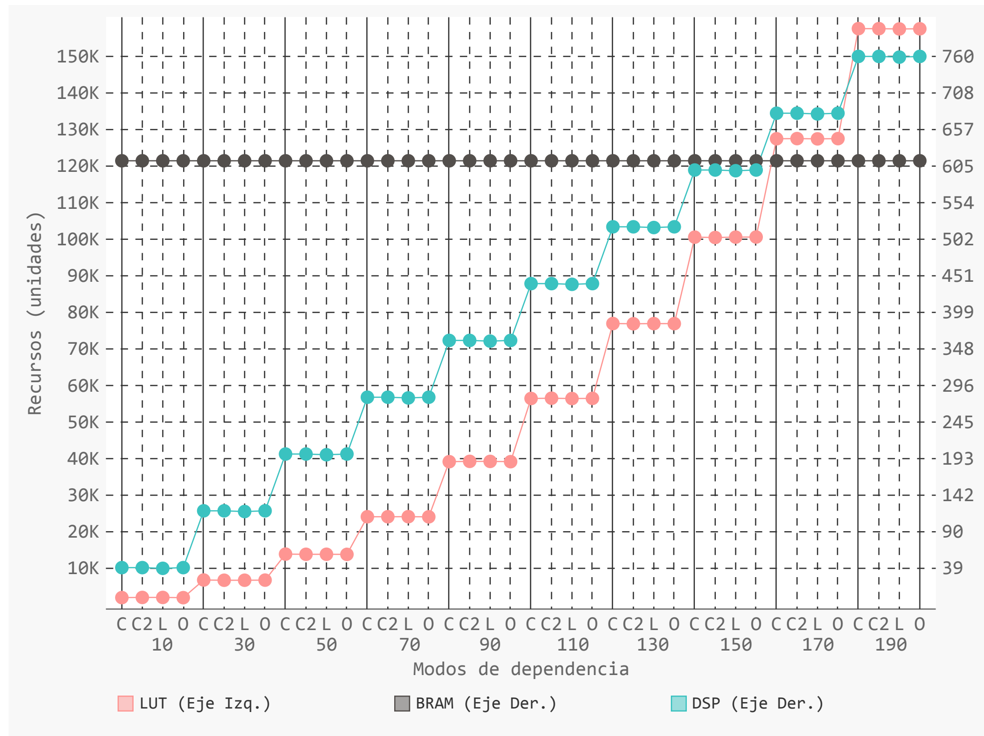


Fig. 5.22: Recursos *hardware* requeridos por la implementación según el modo de dependencia del sistema: circular (C), circular-2 (C2), lineal (L) u opuesto (O). Estos valores son calculados para el rango de sistemas P con tamaños comprendidos entre 10 y 190 reglas. Se representan los recursos LUT (eje izquierdo), BRAM (eje derecho) y DSP (eje derecho).

5.3.2. Recursos *hardware* según complejidad del sistema de entrada

A continuación, se analiza la evolución de los recursos requeridos por Almond PS en función de la variación de la complejidad de los sistemas de entrada, medida en número de reglas. Para ello, se emplea el conjunto de sistemas de prueba y los parámetros fijados en las secciones anteriores.

Respecto al consumo de los recursos BRAM, este es representado gráficamente en la Fig. 5.23). Así, la práctica totalidad son empleados en la interfaz de entrada/salida, propietaria de XILINX. Únicamente un bloque de memoria es empleado por la implementación del sistema P. En consecuencia, la implementación de otra interfaz puede ocasionar un cambio significativo en el uso de este recurso, redu-

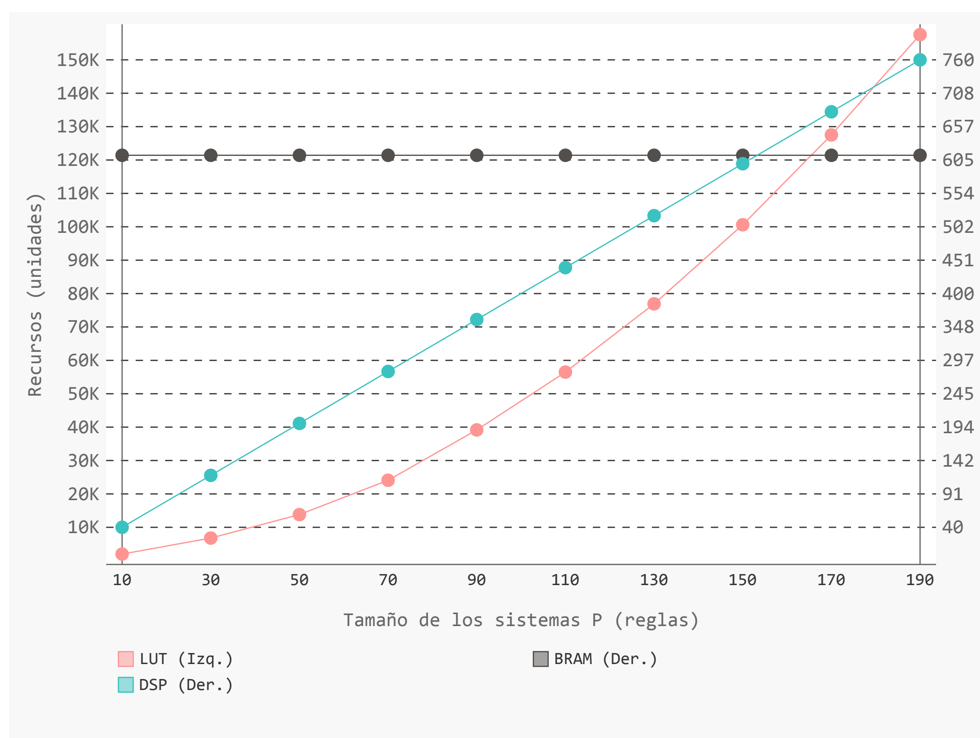


Fig. 5.23: Recursos *hardware* requeridos por la implementación según la complejidad del sistema, medida en número de reglas. LUT usa el eje Y de la izquierda, mientras que BRAM y DSP usan el eje Y de la derecha.

ciéndose a una unidad en caso de que el nuevo bloque no lo requiera.

Atendiendo a los recursos de tipo DSP, estos son empleados en exclusiva por los módulos aritméticos en los que se requieren las operaciones de división y multiplicación. Concretamente, los módulos en los que son utilizados son el Bloque de Cómputo de N_{r_i}, y el Bloque de Asignación. La dependencia de este recurso es lineal respecto al número de reglas, tal y como se muestra en la Fig. 5.23.

Por último, se analizan los recursos de tipo LUT, estos son los más afectados por el aumento de la complejidad, según se muestra en la Fig. 5.23, sufriendo, cuando la implementación se estabiliza, un incremento de orden lineal por cada regla adicional, tal y como se representa en la Fig. 5.24, $y = 3{,}95x + 78{,}5$, que se traduce a un orden cuadrático cuando se toma una referencia absoluta, $y = x(3{,}95x + 78{,}5)$, en la Fig. 5.23.

Dada la gran demanda de este recurso, se considera relevante el estudio de la

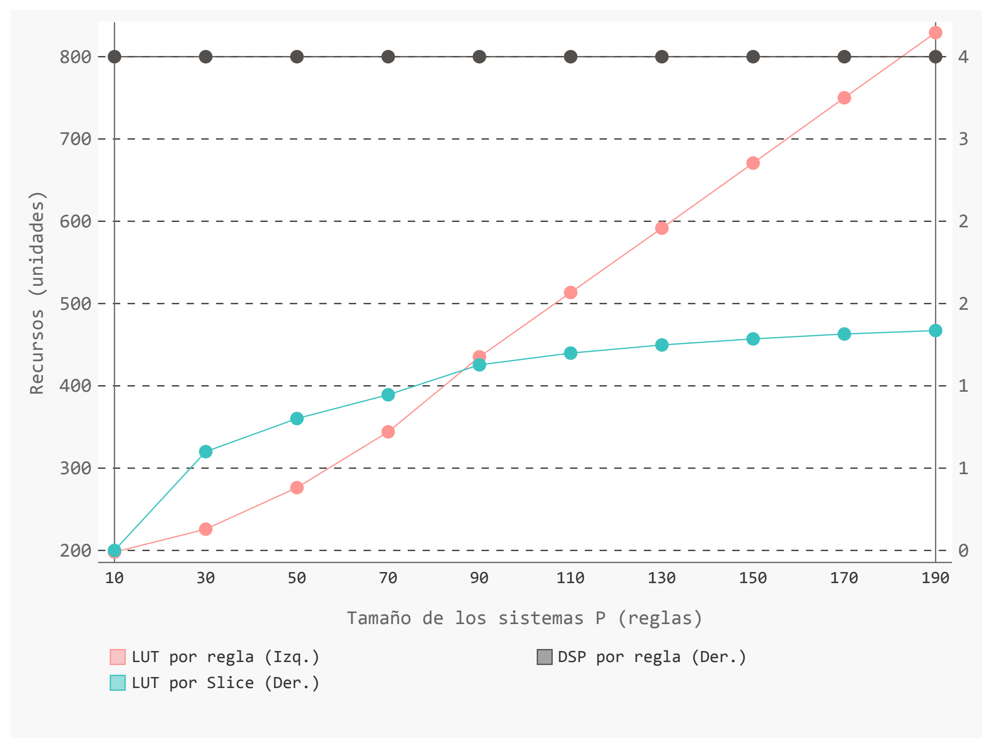


Fig. 5.24: Evolución de los recursos LUT por regla, LSP (LUT por *Slice*) y DSP por regla requeridos por Almond PS según la complejidad de los sistemas, medida en número de reglas. La escala izquierda del eje Y corresponde a los LUT, y la derecha a LPS y DSP.

utilización de los LUT, mostrando los resultados obtenidos en la Fig. 5.25. En este sentido, la gran mayoría son destinados a lógica, siguiendo el orden global. Seguidamente se destinan a rutado, mostrando un crecimiento lineal, y a memoria, en un orden constante. Respecto a la distribución de este recurso entre los distintos componentes de la arquitectura, la Fig. 5.26 recoge como el Bloque de Asignación es el destino de la gran mayoría de los LUT, y el que marca el orden de evolución de este recurso.

En relación a los *slice* consumidos, inicialmente siguen una tendencia similar a la de los LUT, con una relación 1 - 1,5 de *LUT/Slice* para complejidades inferiores a 30. A partir de un tamaño de 90 reglas, las herramientas de XILINX empiezan a distribuir los LUT de un modo más uniforme, hasta alcanzar un ratio de 2,3 *LUT/Slice*.

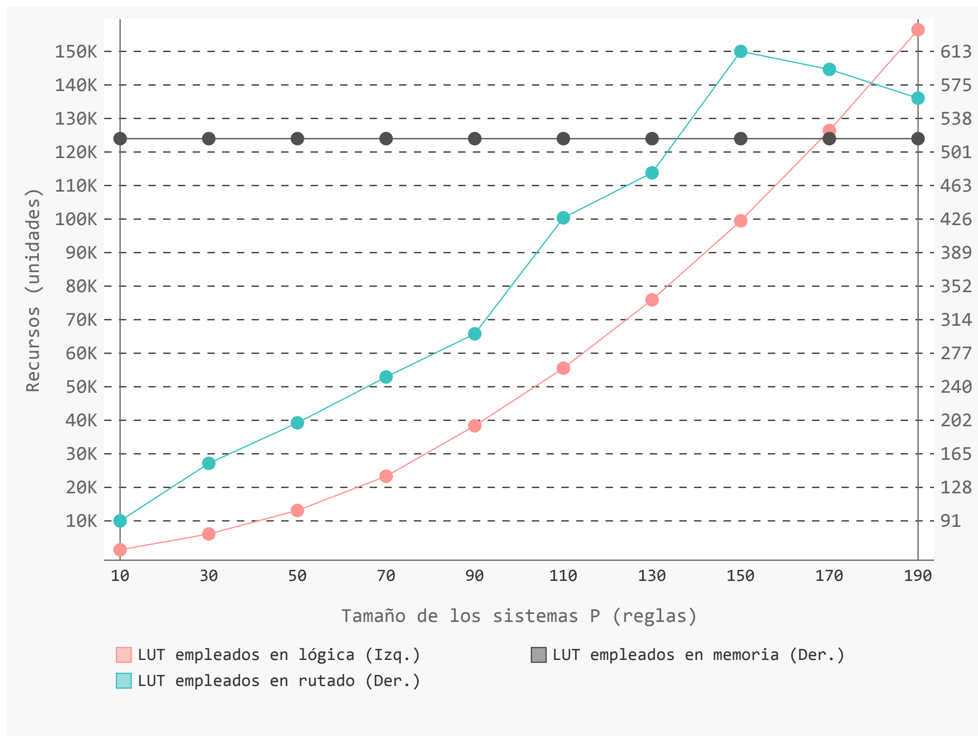


Fig. 5.25: Uso de los recursos LUT, según complejidad del sistema, medida en número de reglas. Los LUT empleados en lógica usa el eje Y de la izquierda, los empleados en rutado y memoria el eje Y de la derecha.

5.3.3. Recursos *hardware* según cardinalidades máximas

El siguiente parámetro a considerar es la cardinalidad máxima de cada objeto soportada por la arquitectura Almond PS. Este parámetro debe ser configurado previamente a la generación de las implementaciones específicas, ya que determina la anchura, en bits, del camino de datos y de las unidades funcionales. Por lo tanto, se ha considerado relevante el análisis de la evolución del sistema respecto a esta magnitud.

La Fig. 5.27 muestra, empleando un gráfico de líneas, los recursos empleados según el tamaño del sistema y los bits considerados. Respecto a los recursos BRAM y DSP, estos no se ven afectados para tamaños de bus inferiores o iguales a 32. En el caso de los DSP, es preciso tener en cuenta que se trata de un recurso *On Chip* específico, y por lo tanto está sujeto a la tecnología de fabricación de la

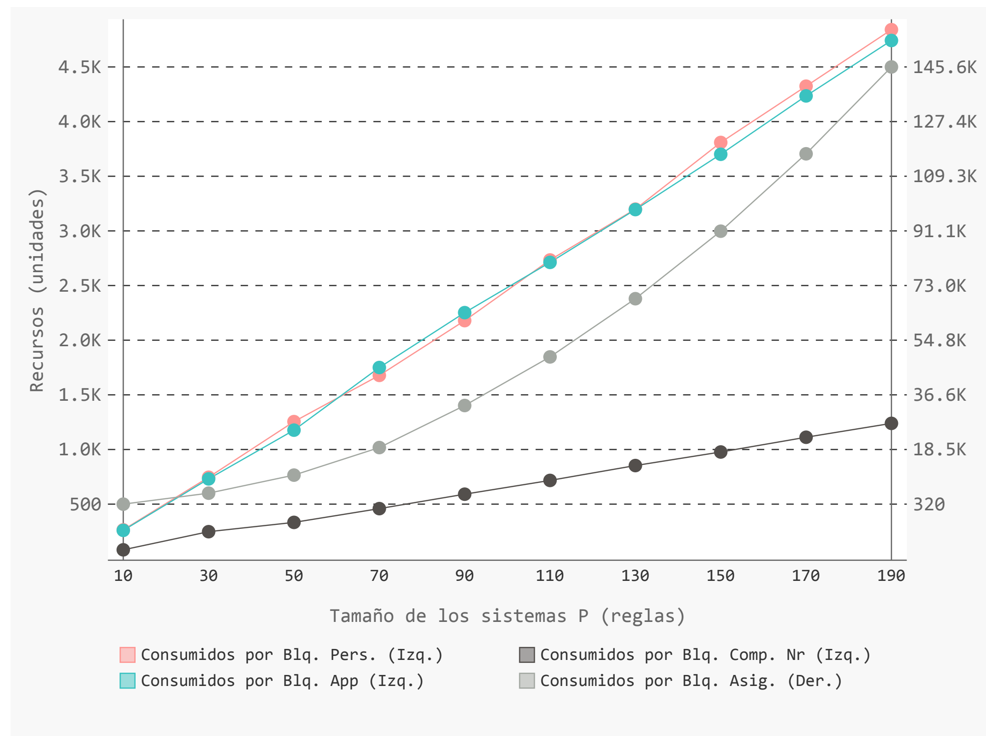


Fig. 5.26: Distribución de los recursos LUT entre los principales módulos de la arquitectura del sistema, según complejidad del sistema, medida en número de reglas. Para el Bloque de Asignación se usa el eje Y de la izquierda, mientras que para el resto se emplea el eje Y de la derecha.

FPGA. En el caso de la *Virtex* 7, el máximo ancho permitido por cada uno de los operandos es de 47 bits. Para tamaños mayores, se debería duplicar el número de este tipo de recursos. Respecto al uso de BRAM, no se considera relevante según la multiplicidad máxima del sistema, ya que son utilizadas para la entrada/salida y para el almacenamiento de constantes. Es por ello que, si bien es probable que para tamaños mayores se requiera un mayor número de recursos, su aumento no es relevante ni crítico para la implementación.

En relación al consumo de *Slice* y LUT, sí es apreciable el impacto de aumentar el tamaño de los registros y buses de la arquitectura. Atendiendo a los LUT, el consumo de recursos es proporcional al incremento de bits: el incremento medio de demanda de recursos al aumentar el tamaño de 16 a 32 bits, es 2,3 veces el demandado al aumentar de 8 a 16 bits. Respecto a los *Slice*, estos son afectados

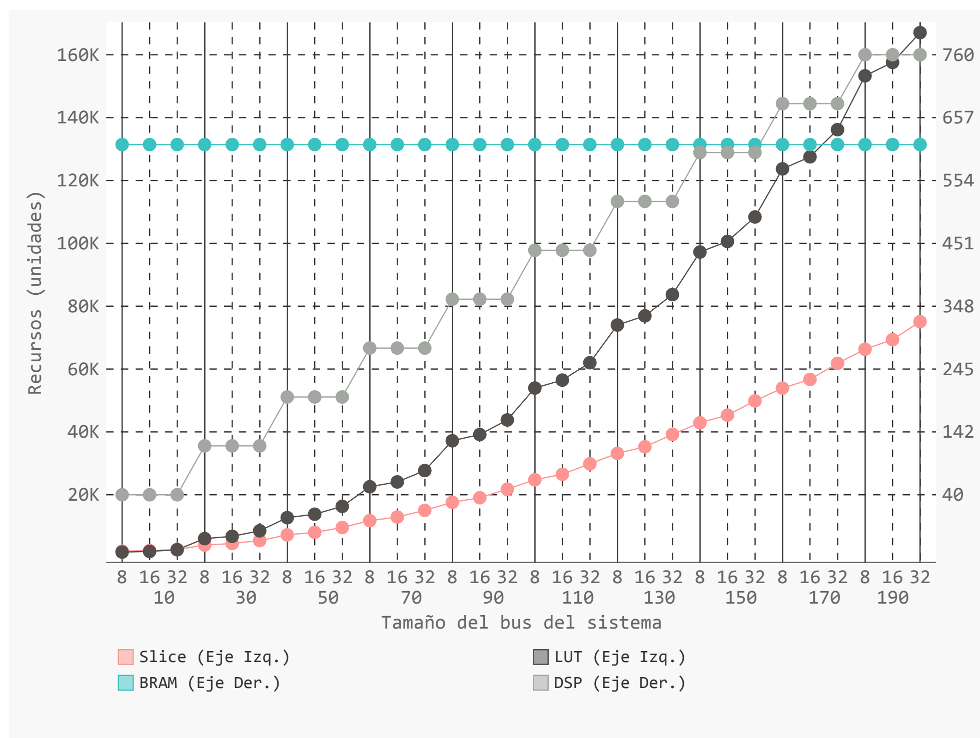


Fig. 5.27: Recursos *hardware* requeridos por la implementación según tamaño de buses y registros, medido en bits. Para los recursos LUT y *Slice* se usa el eje Y de la izquierda, mientras que para BRAM y DSP se usa el eje Y de la derecha.

pero en una proporción menor. Así el incremento medio se sitúa en 1,86 veces al aumentar de 16 bits a 32, respecto al incremento de 8 a 16. En lo que respecta a la relación de LUT por *Slice*, la representación mostrada en la Fig. 5.28 indica que el distinto ritmo de crecimiento hace que, al principio, el ratio sea incluso inferior a la unidad, para posteriormente ir aumentando el uso de LUT por *Slice*. En este sentido, para tamaños superiores a 70 reglas, se observa como la relación de diferencia para las distintas anchuras de bits se estabiliza.

Por último, en la Fig. 5.29 se analiza el impacto por cada uno de los módulos. Así, el Bloque de Persistencia en el que se ve más afectado por la modificación de este parámetro. Este hecho es normal, debido a que es el módulo encargado del almacenamiento de las configuraciones del sistema. Por el contrario, el Bloque de Asignación es el que menos se ve afectado, en orden de magnitud, hasta el punto de que las representaciones de las tres magnitudes se solapan en la representación

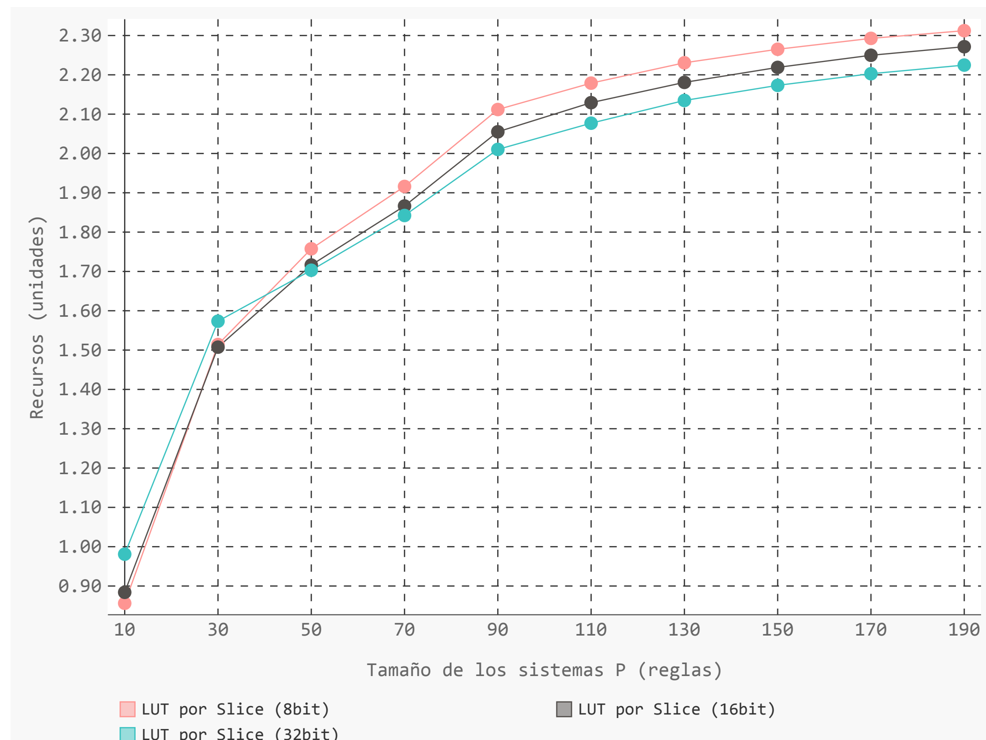


Fig. 5.28: LUT por *Slice* empleados según la anchura de datos empleada. Para cada tamaño considerado, se ofrece la relación, para los tamaños 8, 16 y 32.

gráfica.

En relación con el valor utilizado en el resto de secciones del apartado, se emplea el valor intermedio de 16 bits, al considerarse que mantiene un compromiso entre recursos empleados y capacidad de representación.

5.3.4. Recursos *hardware* según modo de generación *software*

Las herramientas de generación del fabricante permiten fijar parámetros relacionados con la generación del sistema *hardware*. En este sentido, se han establecido tres configuraciones distintas, buscando el equilibrio, favoreciendo el ahorro de recursos y primando la potencia.

Este análisis se representa gráficamente en la Fig. 5.30, donde se muestra en un gráfico de líneas el consumo de *Look-Up Table* (LUT), DSP y BRAM para dis-

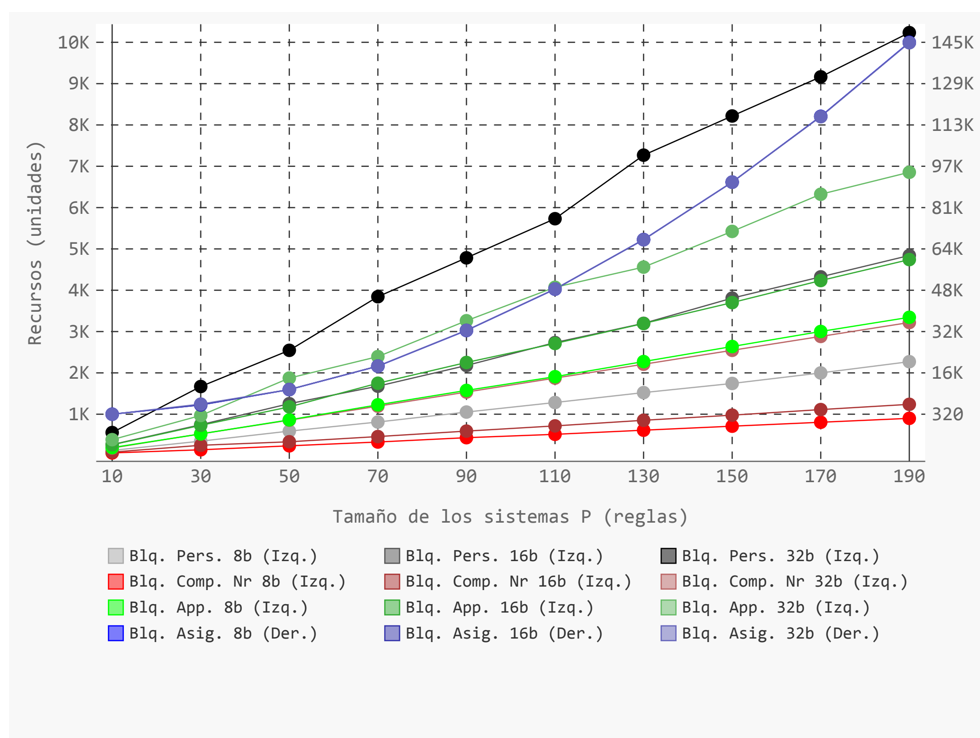


Fig. 5.29: Recursos LUT requeridos por cada uno de los módulos más relevantes de la arquitectura Almond PS, según tamaño de buses y registros, medido en bits. Para el Bloque de Asignación se usa el eje de la derecha, para el resto de los bloques, el de la izquierda. Es preciso destacar qe la representación de los recursos empleados por el Bloque de Asignación (escala de grises) se solapan para las tres cardinalidades consideradas.

tintos tamaños, variando los modos de generación para cada uno de ellos. Según los resultados, los modos de generación no afectan al uso de recursos específicos, los DSP y BRAM. En el caso de los primeros, estos son instanciados de forma directa en el código HDL, por lo que son empleados en la generación. Respecto a los segundos, son usados para el almacenamiento de constantes y, especialmente, por el módulo de entrada/salida, por lo que su uso no se ve afectado por la variación de los parámetros analizados. Respecto a los LUT, son apreciables ligeras diferencias cuando se activa la reducción de recursos. No obstante, las diferencias no son especialmente significativas, por lo que la consideración de un modo u otro es irrelevante o depende de la optimización conseguida en tiempo de procesamiento. En consecuencia, se considera, para el resto de secciones el modo de

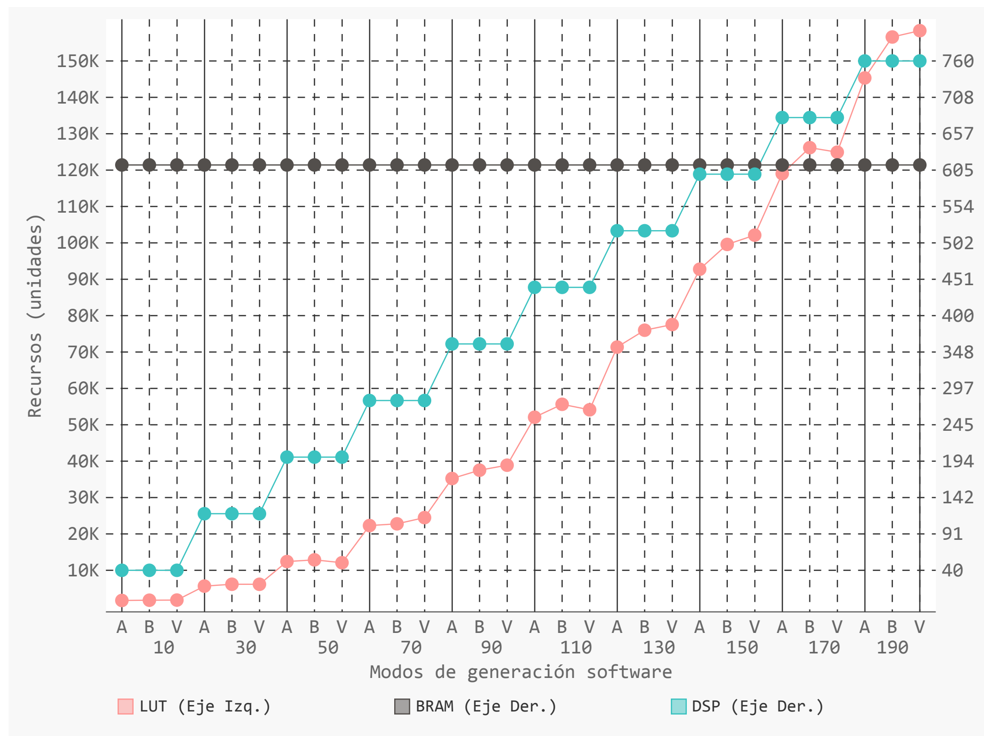


Fig. 5.30: Recursos hardware consumidos por Almond PS según el modo de generación de las herramientas: balanceado (B), primando área (A) y primando velocidad de cómputo V.

generación balanceado, al no existir diferencias notables entre los distintos modos de generación.

5.3.5. Recursos *hardware* según tecnologías de FPGA

El último aspecto a analizar es el impacto de la familia de FPGA. En este sentido, se emplean las tres últimas familias de la compañía XILINX. Concretamente, se han seleccionado aquellas empleadas en las placas de desarrollo que el fabricante destina al programa universitario. Concretamente son:

Familia *Virtex5* : *Virtex-5 FX70T* empleada en la placa *ML507*.

Familia *Virtex6* : *Virtex-6 LX240T* empleada en la placa *ML605*.

Familia *Virtex7* : *Virtex-7 VX485T* empleada en la placa *VC707*.

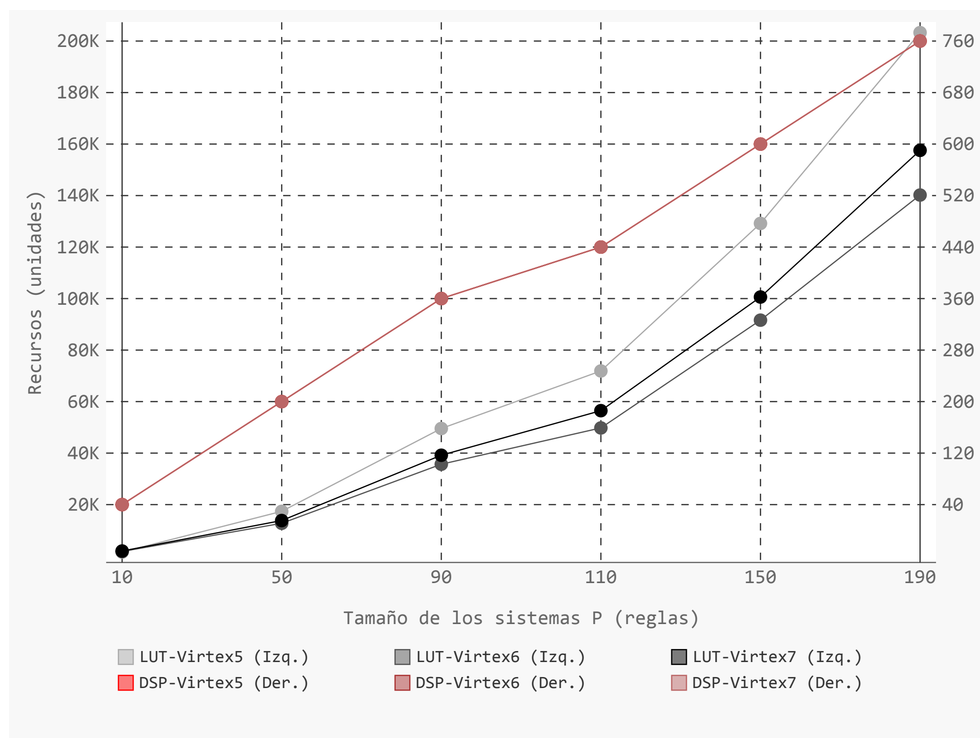


Fig. 5.31: Recursos *hardware* requeridos por la implementación según las tres últimas series de FPGA del fabricante XILINX. LUT usa el eje Y de la izquierda, mientras que DSP usa el eje Y de la derecha. Los LUT empleados por las tres familias son iguales, por lo que su representación está solapada.

En la Fig 5.31 se presentan los resultados. En la gráfica se muestran valores para LUT y DSP, los recursos BRAM dependen de los recursos empleados por la interfaz de entrada/salida, tal y como se refiere en la Sección 4.2.2 (Pág. 135), por lo que no se muestran. En primer lugar, los dispositivos DSP dedicados son equivalentes en las tres familias, por lo que no se aprecia cambio alguno en su consumo. Respecto al consumo de LUT, se aprecia como para las familias *Virtex6* y *Virtex7* se obtienen resultados similares, mientras que para el modelo *Virtex5*, se consume un mayor número de recursos, aunque siempre en el mismo orden de magnitud. Este aumento es motivado por la falta de recursos DSP dedicados, que son implementados en lógica programanble.

Por último, sí es preciso destacar que en la FPGA *Virtex-5 FX70T*, no existen suficientes recursos de tipo DSP para los resultados mostrados, por lo que es

conveniente el empleo de divisores implementados en lógica. Esto es necesario a partir de 110 reglas, inclusive. En este sentido, en la Fig. 5.31 se ha representado el número de DSP necesario, independientemente de la disponibilidad de recursos en la FPGA. Respecto a la FPGA *Virtex-6 LX240T*, 210 reglas es el límite del tamaño de entrada que puede ser implementado en este dispositivo.

En conclusión, se aprecia que, para las últimas tres familias del fabricante XILINX, el cambio de familia únicamente tiene impacto en el número de recursos disponibles, que aumenta de una familia a otra. En este sentido, el modelo *Virtex-7 VX485T* será el empleado para la adquisición de los resultados en el resto de las secciones, al tratarse del modelo más reciente.

5.4. Análisis de rendimiento

El estudio de la arquitectura Almond PS finaliza con el análisis de rendimiento. En este sentido, se contemplan los sistemas P ya definidos anteriormente. En base a las conclusiones del anterior apartado, se consideran los tamaños comprendidos entre 10 y 190 reglas, con saltos de 20 reglas, dependencia circular, bus de 16 bits y haciendo uso de la FPGA *Virtex-7 VX485T*. Además, también se ofrece una comparación de rendimiento respecto a P-Lingua.

En la Fig. 5.32 se muestra el rendimiento ofrecido por Almond PS. Así, para cada modo de generación *software*, se representan los resultados empleando un gráfico de líneas. Como es esperable, el modo de generación *Velocidad* es el que consigue frecuencias más altas, al primar la velocidad frente al consumo de recursos. Del mismo modo, se observa como las frecuencias alcanzadas por el modo *Balanceado* y *Área* presentan resultados similares en este apartado. Independientemente del modo de generación, en todos los sistemas se han alcanzado los 100 MHz, hasta alcanzar un máximo de 136 MHz para el tamaño de 10 reglas cuando se aplica la optimización *Velocidad*. Para el resto de tamaños, el máximo se sitúa en torno a los 124 MHz.

Por último, en la Fig. 5.33 se muestra el tiempo de ejecución (medido en ms) de la ejecución de los sistemas de prueba en Almond PS, considerando el modo de generación balanceado, así como en P-Lingua. En este punto, se ha obtenido un rendimiento variable, en función del número de reglas, empezando con un factor

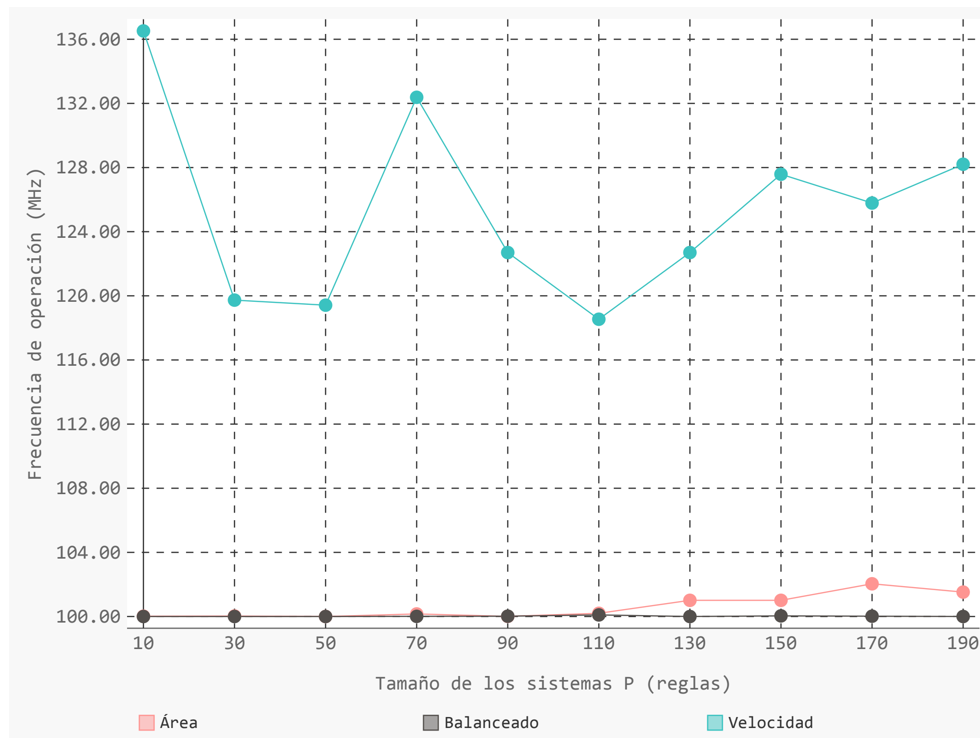


Fig. 5.32: Rendimiento (medido en MHz) de la arquitectura Almond PS, en función de la complejidad del sistema (medida en número de reglas).

de multiplicativo de 2,000 veces, para un tamaño de 10 reglas, hasta alcanzar un factor de 11,000 veces, para un tamaño de 190 reglas, suponiendo una mejora de hasta 4 órdenes de magnitud.

5.5. Conclusiones

En este capítulo se han analizado los resultados funcionales y no funcionales obtenidos con la arquitectura Almond PS. Así, tras plantear los sistemas P de prueba, se ha estudiado el correcto comportamiento de los mismos. A continuación, se han analizado los recursos empleados en base a 5 puntos de variación, entre los que se incluye la variación de la complejidad del sistema. Por último, se ha medido el rendimiento en frecuencia y tiempo de ejecución, comparándose este último con P-Lingua.

Desde un punto de vista funcional, Almond PS consigue resultados válidos. Para

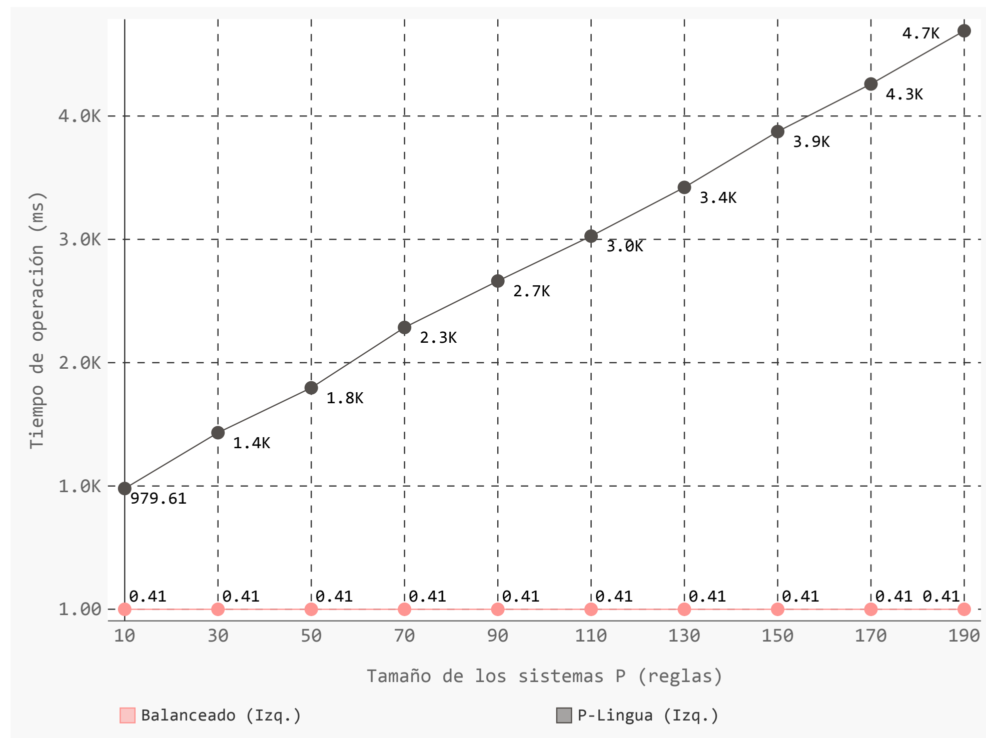


Fig. 5.33: Tiempo de ejecución (medido en ms) de la arquitectura Almond PS, en función de la complejidad del sistema (medida en número de reglas). También se muestran los resultados adquiridos con P-Lingua, así como el número de veces en que Almond PS es más rápido que P-Lingua.

ello, estos se han contrastado con los resultados más probables en base a los cuatro patrones de comportamiento seleccionados para los sistemas de prueba. Del mismo modo, se ha comparado su funcionamiento con los obtenidos por implementaciones *software* usando la herramienta P-Lingua, comprobando que son equivalentes a nivel global.

Atendiendo al consumo de recursos, el empleo de recursos BRAM no constituye una limitación en la implementación de los distintos sistemas. El número de DSP existentes en las FPGA también son los suficientes para los tamaños soportados, comprometiendo la generación únicamente en aquellos casos en los que se empleen multiplicidades máximas que requieran de un número mayor de 47 bits por objeto. Son los recursos de tipo LUT los que limitan la implementación de sistemas P, al obtenerse un consumo exponencial en relación al tamaño del sistema

a implementar, medido en número de reglas.

Respecto a los 5 puntos de variación empleados para la obtención de los resultados, el tamaño del sistema de entrada es el factor que más influye en el consumo de recursos, tal y como se ha comentado en el anterior párrafo, seguido por las cardinalidades máximas. La dependencia entre las reglas y el modo de generación *software* no muestran diferencias significativas en el consumo de recursos. Finalmente, las tecnologías de FPGA comparadas, las tres últimas del fabricante XILINX, no suponen un impacto en el consumo de recursos *hardware*, si bien sí son importantes, debido a que los recursos disponibles aumentan conforme más reciente es la familia.

Por último, se analiza el tiempo de computación. En este sentido, los resultados muestran como Almond PS consigue una optimización considerable en tiempo de ejecución respecto a la implementación *software* P-Lingua, situándose como un candidato para ejecuciones computacionalmente costosas. Al contrastar estos tiempos con el consumo de recursos, los resultados ofrecidos son los esperados, el mantenimiento de un tiempo constante de ejecución tiene su contrapartida en dicho consumo, siendo exponencial para el caso de los de tipo LUT. De ese modo, para un tiempo constante de ejecución se obtiene un consumo de recursos exponencial.

En conclusión, todos los resultados indican que la arquitectura Almond PS es apta para ser empleada en entornos de producción, obteniendo incrementos significativos de rendimiento sin desviación de los resultados. Como único punto, se deja abierta la posibilidad de paliar el impacto de los recursos mediante optimizaciones o combinación con otras implementaciones.

Conclusiones finales

A continuación se resumen las conclusiones más relevantes de este trabajo:

- Se ha llevado a cabo un análisis del modelo de computación con membranas, concluyendo con una descripción estructural y funcional abstracta, basada en el *framework* de Red de Células existente, con la finalidad de establecer los requisitos a satisfacer por cualquier implementación de este tipo de sistemas.

- Se ha desarrollado un algoritmo basado en la teoría de series formales de potencias y gramáticas libres de contexto. Este nuevo enfoque, no aplicado anteriormente en la implementación *hardware* de sistemas P, permite un mejor aprovechamiento de las capacidades de la tecnología FPGA, siendo capaz de realizar una transición en tan solo 5 ciclos de reloj, un valor cercano al ideal de 1 ciclo.

- Se ha diseñado una arquitectura, denominada Almond PS, que hace uso de los algoritmos anteriormente mencionados, presentando las siguientes características:

  - Se trata de una arquitectura con un alto índice de flexibilidad, permitiendo la mejora de cada uno de sus bloques por separado. Además, se ha encapsulado la lógica necesaria para la introducción de nuevas relaciones entre reglas en un único bloque, minimizando el impacto de soportar un mayor rango de sistemas P.
  - Las implementaciones concretas alcanzan un rendimiento de $2 * 10^7$ transiciones por segundo, situándose entre las implementaciones con mayor potencia computacional de la disciplina.

  - Presenta una escalabilidad adecuada en base a sus características de flexibilidad y rendimiento. En este sentido, la modularidad de la arquitectura facilita futuras extensiones y mejoras de la arquitectura.
  - El rango de sistemas P aceptados por Almond PS únicamente se ve afectado por las dependencias entre sus reglas. Es, a diferencia del resto de implementaciones existentes, independiente de los elementos estructurales de estos sistemas: el tipo de reglas y objetos.

- Se ha desarrollado un *software* de generación que permite crear instancias concretas en base a sistemas P de entrada. Además, se han definido las bases para el desarrollo de un *framework* de generación de *hardware*.

- Se ha realizado una propuesta de metodología para la introducción de pruebas unitarias automatizadas en el diseño *hardware*, acompañada de una primera implementación de las herramientas base.

- Se ha realizado un exhaustivo análisis de los requisitos funcionales de Almond PS, incluyendo la comparación con implementaciones de referencia, así como de los requisitos no funcionales y consumo de recursos *hardware*, en base a los distintos parámetros candidatos a influir en estos.

En conclusión, se ha desarrollado una arquitectura *hardware* aplicando un enfoque novedoso en el área, y que destaca por una elevada flexibilidad y potencia computacional. Este trabajo supone, en opinión del doctorando, un claro avance en la implementación *hardware* de este tipo de sistemas. Además, el *software* presentado constituye un punto de partida para el desarrollo de nuevas metodologías y herramientas de trabajo *hardware*.

Publicaciones

Proyectos de investigación

Las tareas llevadas a cabo durante la realización de este trabajo se enmarcan dentro del siguiente proyecto de investigación:

- HIPERSYS: Optimización de Sistemas Empotrados de Altas Prestaciones (MEC TEC-2011-27936).

Resultados directos

La realización de este trabajo ha dado lugar de manera directa a las siguientes publicaciones:

- J. Quiros, S. Verlan, J. Viejo, A. Millan, M. J. Bellido. Fast Hardware Implementations of Static P systems. *Computing and Informatics* (factor de impacto 0,504). [Aceptado. Pendiente de publicación.]

- S. Verlan, J. Quiros (2013). Fast Hardware Implementations of P Systems. *Membrane Computing*, volumen 7762 de *Lecture Notes in Computer Science*, páginas 404- 423. Springer Berlin / Heidelberg. ISBN: 978-3-642-36750-2.

Resultados indirectos

Además de las publicaciones citadas, algunas de las tareas llevadas a cabo durante la realización de este trabajo han supuesto una aportación importante en las siguientes publicaciones:

- J. Quiros, J. Viejo, A. Millan, A. Muñoz, J. I. Villar, D. Guerrero (April 2011) Implementation of a configuration server for a hardware SNTP synchronization platform based on FPGA. *In Proc. 7th Southern Conference on Programmable Logic (SPL)*. pp. 239–244. Cordoba (Argentina). ISBN: 978-1-4244-8846-9.

- J. Quiros, J. Viejo, A. Muñoz, A. Millan, E. Ostua, J. I. Villar (February 2010) Implementación sobre FPGA de un cliente SNTP usando MicroBlaze. *In Proc. 16th Iberchip Workshop (IWS)*. pp. –. Iguazu Falls (Brazil).

- J. Viejo, J. I. Villar, J. Juan, A. Millan, E. Ostua, J. Quiros (2012) Long-term on-chip verification of systems with logical events scattered in time. *Microprocessors and Microsystems 36 (5)* pp. 402–408. Elsevier. United Kingdom. ISSN: 0141-9331.

Abreviaturas

ACS *Artificial Cell Systems* 53

ADN Ácido desoxirribonucleico 18

ALU *Arithmetic Logic Unit* 136

ASIC *Application-Specific Integrated Circuit* 21

AST *Abstract Syntax Tree* 130

BBB *Binomial Block Based Algorithm* 55

BRAM *Block RAM* 22

CLB *Configurable Logic Blocks* 22

CIM *Computation Independent Model* 138

CPLD *Complex Programmable Logic Device* 22

CPS *Conversational P Systems* 50

CPU *Central Processing Unit* 55

CSPS *Client-Server P System* 57

CSV *Comma-Separated Values* 148

CUDA *Compute Unified Device Architecture* 20

DCM *Digital Clock Manager*

DCBA *Direct distribution based on Consistent Blocks Algorithm* 55

DNDP *Direct Non-Deterministic distribution with Probabilities* 55

DMPS *Dynamic Meaning P Systems* 50

DND *Direct Nondeterministic Distribution* 68

DSP *Digital Signal Processor* 190

EMF *Eclipse Modeling Framework* 139

EGL *Epsilon Generation Language* 141

EOL *Epsilon Object Language* 140

ETL *Epsilon Transformation Language* 141

EVL *Epsilon Validation Language* 141

FF *Flip Flops*

FPGA *Field Programmable Gate Array* 7

GPU *Graphic Processor Unit* 20

HDL *Hardware Description Language* 26

ICO *ChipScope Integrated Controller* 120

IDE *Integrated Development Environment* 130

ILA *Integrated Logic Analyzer* 120

IOB *Input/Output Blocks* 22

LFSR *Linear Feedback Shift Register* 110

LPS *Linguistic P System* 50

LUT *Look-Up Table* 14

MDA *Model-Driven Architecture* 138

MDE *Model-Driven Engineering* 6

MOF *OMG's MetaObject Facility* 139

MPI *Message Passing Interface* 57

NPQ *Non Photochemical Quenching* 49

OMG *Object Management Group* 138

PCI Peripheral Component Interconnect

PDP Sistemas P de Dinámica de Poblaciones 55

PIM *Platform Independent Model* 138

PSM *Platform Specific Model* 138

RGNC Grupo de Computación Natural de la Universidad de Sevilla 53

RMI *Remote Method Indication* 57

SBML *Systems Biology Markup Language* 53

SIMT *Single Instruction Multi-Thread*

SOC *System On Chip* 138

SPMD *Single Program Multiple Data* 20

UML *Unified Modeling Language* 11

UPM Universidad Politécnica de Madrid 53

VIO *Virtual Input/Output* 120

XMI *XML Metadata Interchange* 139

XML *eXtensible Markup Language* 139

Bibliografía

(2006). Extensible markup language (xml) 1.1 (second edition). 139

Alhazov, A., Boian, E., Cojocaru, S., and Rogozhin, Y. (2009). Modelling Inflections in Romanian Language by P Systems with String Replication. *The Computer Science Journal of Moldova*, 17:160–178. 50

Alhazov, A. and Sburlan, D. (2006). Static Sorting P Systems. In Ciobanu, G., Păun, G., and Pérez-Jiménez, M. J., editors, *Applications of Membrane Computing*, chapter 8, pages 215–252. Springer. 49

Alonso, S., Fernández, L., Arroyo, F., and Gil, F. J. (2008). A Circuit Implementing Massive Parallelism in Transition P Systems. *Institute of Information Theories and Applications FOI ITHEA*, 2(1). 64

Aman, B. and Ciobanu, G. (2008). Membrane systems with surface objects. In *International Workshop on Computing with Biomolecules (CBM)*, pages 17–28, Vienna, Austria. 36

Aman, B. and Ciobanu, G. (2009). Mutual Mobile Membranes Systems with Surface Objects. In *Seventh Brainstorming Week on Membrane Computing*, volume 1, pages 29–39, Seville, Spain. 36

Andrei, P. and Mario, J. P. (2006). Modelling Signal Transduction using P Systems. In Hoogeboom, H., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 4361 of *Lecture Notes in Computer Science*, pages 100–122. Springer Berlin / Heidelberg. 49

Ardelean, I. I., Besozzi, D., Garzon, M. H., Mauri, G., and Roy, S. (2006). P System Models for Mechanosensitive Channels. In *Applications of Membrane Computing*, chapter 2, pages 43–81. Springer. 49

Arroyo, F., Baranda, A. V., Castellanos, J., Luengo, C., and Mingo, L. F. (2001a). A Recursive Algorithm for Describing Evolution in Transition P Systems. In *Pre-Proccedings of Workshop on Membrane Computing (WMC-C de A2001)*, pages 19–30. 53

Arroyo, F., Baranda, A. V., Castellanos, J., Luengo, C., and Mingo, L. F. (2001b). Structures and Bio-language to Simulate Transition P Systems on Digital Computers. In Calude, C., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Multiset Processing*, volume 2235 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg. 53

Arroyo, F., Castellanos, J., Luengo, C., and Mingo, L. F. (2004a). A Binary Data Structure for Membrane Processors : Connectivity Arrays. In Martín-Vide, C., Mauri, G., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 19–30. Springer Berlin / Heidelberg, Tarragona, Spain. 61

Arroyo, F., Luengo, C., Baranda, A. V., and Mingo, L. D. (2003). A Software Simulation of Transition P Systems in Haskell. In Păun, G., Rozenberg, G., Salomaa, A., and Zandron, C., editors, *Membrane Computing*, volume 2597 of *Lecture Notes in Computer Science*, pages 19–32. Springer Berlin / Heidelberg. 53

Arroyo, F., Luengo, C., Castellanos, J., and Mingo, L. F. (2004b). Representing Multisets and Evolution Rules in Membrane Processors. In *Fifth Workshop on Membrane Computing (WMC5)*, pages 126–137, Milano, Italy. 61

Atanasiu, A. (2003). Authentication of messages using P systems. In Păun, G., Rozenberg, G., Salomaa, A., and Zandron, C., editors, *Membrane Computing*, volume 2597 of *Lecture Notes in Computer Science*, pages 33–42. Springer Berlin / Heidelberg. 50

Balbotín-Noval, D., Pérez-Jiménez, M. J., and Sancho-Caparrini, F. (2003). A MzScheme Implementation of Transition P Systems. In Păun, G., Rozenberg, G., Salomaa, A., and Zandron, C., editors, *Membrane Computing*, volume 2597 of *Lecture Notes in Computer Science*, pages 58–73. Springer Berlin / Heidelberg. 53

Baranda, A. V., Arroyo, F., Castellanos, J., and Gonzalo, R. (2002). Towards an Electronic Implementation of Membrane Computing : A Formal Description of Non-deterministic Evolution in Transition P Systems. In Jonoska, N. and Seeman, N., editors, *Membrane Computing*, volume 2340 of *Lecture Notes in Computer Science*, pages 350–359. Springer Berlin / Heidelberg. 53

Baranda, A. V., Castellanos, J., Gonzalo, R., Arroyo, F., and Mingo, L. F. (2001). Data Structures for Implementing Transition P Systems in Silico. *Romanian Journal of Information Science and Technology*, 4:21–32. 53

Bernardini, F. and Gheorghe, M. (2004). Cell Communication in Tissue P Systems and Cell Division in Population P Systems. In *Second Brainstorming Week on Membrane Computing*, pages 74–91, Seville, Spain. 36

Besozzi, D. and Ciobanu, G. (2005). AP system description of the sodium-potassium pump. In Mauri, G., Păun, G., Pérez-Jiménez, M. J., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 3365 of *Lecture Notes in Computer Science*, pages 210–223. Springer Berlin / Heidelberg. 49

Bianco, L. and Castellini, A. (2007). Psim : A Computational Platform for Metabolic P Systems. In Eleftherakis, G., Kefalas, P., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg. 54

Bianco, L., Fontana, F., Franco, G., and Manca, V. (2006). P Systems for Biological Dynamics. In Ciobanu, G., Păun, G., and Pérez-Jiménez, M. J., editors, *Applications of Membrane Computing*, Natural Computing Series, chapter 3, pages 83–128. Springer Berlin / Heidelberg. 54

Bottoni, P., Martín-Vide, C., Păun, G., and Rozenberg, G. (2002). Membrane systems with promoters/inhibitors. *Acta Informatica*, 38:695–720. 35

Cabarle, F., Adorna, H., and Martínez-del Amor, M. A. (2011a). A Spiking Neural P system simulator based on CUDA. In *Twelveth Conference on Membrane Computing*, pages 77–92, Fontainebleau, France. 56

Cabarle, F., Adorna, H., and Martínez-del Amor, M. A. (2011b). An Improved GPU Simulator For Spiking Neural P Systems. In *IEEE Sixth International Conference on Bio-Inspired Computing: Theories and Applications*, Penang, Malaysia. 56

Cabarle, F., Adorna, H., and Martínez-del Amor, M. A. (2011c). Simulating Spiking Neural P Systems Without Delays Using GPUs. In *Ninth Brainstorming Week on Membrane Computing*, pages 23–42, Seville, Spain. Fénix Editora. 56

Cardona, M., Colomer, M. A., Margalida, A., Palau, A., Pérez-Hurtado, I., Pérez-Jiménez, M. J., and Sanuy, D. (2010). A computational modeling for real ecosystems based on P systems. *Natural Computing*, 10(1):39–53. 48, 55, 79, 87, 96

Cardona, M., Colomer, M. A., Pérez-Jiménez, M. J., Sanuy, D., and Margalida, A. (2008). A P System Modeling an Ecosystem Related to the Bearded Vulture. In *Sixth Brainstorming Week on Membrane Computing*, pages 51–66, Seville, Spain. 48

Cardona, M., Colomer, M. A., Pérez-Jiménez, M. J., Sanuy, D., and Margalida, A. (2009). Modelling ecosystem using P systems: The bearded vulture, a case study. In Corne, D. W., Frisco, P., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 137–156. Springer Berlin / Heidelberg. 48

Carlos Martín-Vide, Păun, G., Pazos, J., and Rodríguez-Patón, A. (2005). Tissue P systems with channel states. *Theoretical Computer Science*, 296(2):295–326. 36

Castellini, A. and Franco, G. (2008). On Modeling Signal Transduction Networks. In *Sixth Brainstorming Week on Membrane Computing*, pages 67–77, Seville, Spain. 49

Castellini, A. and Manca, V. (2009). MetaPlab : A Computational Framework for Metabolic P Systems. In Corne, D. W., Frisco, P., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 157–168. Springer Berlin / Heidelberg. 54

Cavaliere, M. and Ardelean, I. I. (2006). Modeling Respiration in Bacteria and Respiration/Photosynthesis Interaction in Cyanobacteria Using a P System Simulator. In *Applications of Membrane Computing*, chapter 4, pages 129–158. Springer. 49

Cazzaniga, P., Pescini, D., Romero-Campero, F. J., Besozzi, D., and Mauri, G. (2006). Stochastic Approaches in P Systems for Simulating Biological Systems. In *Fourth Brainstorming Week on Membrane Computing*, pages 145–164, Seville, Spain. 49

Cecilia, J. M., García, J. M., Guerrero, G. D., Martínez-del Amor, M. a., Pérez-Hurtado, I., and Pérez-Jiménez, M. J. (2009a). Simulation of P systems with active membranes on CUDA. *Briefings in bioinformatics*, 2(3):313–22. 55

Cecilia, J. M., García, J. M., Guerrero, G. D., Martínez-del Amor, M. A., Pérez-Hurtado, I., and Pérez-Jiménez, M. J. (2010a). Implementing P Systems Parallelism by Means of GPUs. In Păun, G., Pérez-Jiménez, M. J., Riscos-Núñez, A., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 227–241. Springer Berlin / Heidelberg. 55

Cecilia, J. M., García, J. M., Guerrero, G. D., Martínez-del Amor, M. A., Pérez-Hurtado, I., and Pérez-Jiménez, M. J. (2010b). Simulating a P system based efficient solution to SAT by using GPUs. *Journal of Logic and Algebraic Programming*, 79(6):317–325. 55

Cecilia, J. M., Guerrero, G. D., García, J. M., Martínez-del Amor, M. A., Pérez-Hurtado, I., and Pérez-Jiménez, M. J. (2009b). Simulation of P Systems with Active Membranes on CUDA. In *International Workshop on High Performance Computational Systems Biology, 2009. HIBI '09.*, pages 61–70, Trento, Italy. IEEE Computer Society. 55

Chomsky, N. and Schützenberger, M.-P. (1963). The Algebraic Theory of Context-Free Languages. In *Computer Programming and Formal Systems*, pages 118–161. North Holland. 82, 84

Christinal, H. A., Díaz-Pernil, D., Gutiérrez-Naranjo, M. A., and Pérez-Jiménez, M. J. (2010). Array Tissue-like P Systems. In *Eighth Brainstorming Week on Membrane Computing*, pages 37–51, Seville, Spain. 50

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363. 17

Ciobanu, G. (2003). Distributed algorithms over communicating membrane systems. *Bio Systems*, 70(2):123–133. 56, 57

Ciobanu, G. (2006). Modeling Cell-Mediated Immunity by Means of P Systems. In *Applications of Membrane Computing*, chapter 5, pages 159–180. Springer. 48

Ciobanu, G., Desai, R., and Kumar, A. (2003a). Membrane Systems and Distributed Computing. In Păun, G., Rozenberg, G., Salomaa, A., and Zandron, C., editors, *Membrane Computing*, volume 2597 of *Lecture Notes in Computer Science*, pages 187–202. Springer Berlin / Heidelberg. 56, 57

Ciobanu, G., Dumitriu, D., Huzum, D., Moruz, G., and Tanasa, B. (2003b). Client – Server P Systems in Modeling Molecular Interaction. In Păun, G., Rozenberg, G., Salomaa, A., and Zandron, C., editors, *Membrane Computing*, volume 2597 of *Lecture Notes in Computer Science*, pages 203–218. Springer Berlin / Heidelberg. 56, 57

Ciobanu, G. and Guo, W. (2004). P Systems Running on a Cluster of Computers. In Martín-Vide, C., Mauri, G., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 123–139. Springer Berlin / Heidelberg. 56, 57

Ciobanu, G. and Paraschiv, D. (2002). P system software simulator. *Fundamenta Informaticae*, 49:61–66. 56, 57

Colomer, M., Martínez-del Amor, M. A., Pérez-Hurtado, I., Perez-Jimenez, M., and Riscos-Nuñez, A. (2010). A uniform framework for modeling based on p systems. In *Bio-Inspired Computing: Theories and Applications (BIC-TA), 2010 IEEE Fifth International Conference on*, pages 616–621. 55, 79, 96

Colomer, M. A., Lavín, S., Marco, I., Margalida, A., Pérez-Hurtado, I., Pérez-Jiménez, M. J., Sanuy, D., Serrano, E., and Valencia-Cabrera, L. (2011a). Modeling Population Growth of Pyrenean Chamois (Rupicapra p. pyrenaica) by using P systems. In Gheorghe, M., Hinze, T., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 144–159. Springer Berlin / Heidelberg. 48

Colomer, M. A., Margalida, A., Sanuy, D., and Pérez-Jiménez, M. J. (2011b). A bio-inspired computing model as a new tool for modeling ecosystems: The avian scavengers as a case study. *Ecological Modelling*, 222(1):33–47. 48

Colomer, M. A., Montori, A., Gaspa, I., and Fondevilla, C. (2011c). A computational model to study the dynamics of Pyrenean Newt (Calotriton asper). In *Twelveth Conference on Membrane Computing*, Fontainebleau, France. 48

Cordón-Franco, A., Gutiérrez-Naranjo, M. A., Pérez-Jiménez, M. J., and Sancho-Caparrini, F. (2004). A Prolog simulator for deterministic P systems with active membranes. *New Generation Computing*, 22(4):349–363. 53

Cordón-Franco, A. and Sancho-Caparrini, F. (2005). Approximating Non-discrete P Systems. In Mauri, G., Păun, G., Pérez-Jiménez, M. J., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 3365 of *Lecture Notes in Computer Science*, pages 287–295. Springer Berlin / Heidelberg. 48

Daniel, D.-P., Gutiérrez-Naranjo, M. A., Real, P., and Sánchez-Canales, V. (2010). A Cellular Way to Obtain Homology Groups in Binary 2D Images. In *Eighth Brainstorming Week on Membrane Computing*, pages 89–99, Seville, Spain. 50

Dassow, J. and Păun, G. (2001). P systems with Communication Based on Concentration. *Acta Cybernetica*, 15:9–24. 35

Díaz-Pernil, D., Pérez-Hurtado, I., Pérez-Jiménez, M. J., and Riscos-Núñez, A. (2008). P-Lingua : A Programming Language for Membrane Computing. In *Sixth Brainstorming Week on Membrane Computing*, pages 135–155, Seville, Spain. 53

Díaz-Pernil, D., Pérez-Hurtado, I., Pérez-Jiménez, M. J., and Riscos-Núñez, A. (2009). A P-Lingua Programming Environment for Membrane Computing. In Corne, D. W., Frisco, P., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 187–203. Springer Berlin / Heidelberg. 53

Enguix, G. B. (2004). Preliminaries about Some Possible Applications of P Systems in Linguistics. In Martín-Vide, C., Mauri, G., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 74–89. Springer Berlin / Heidelberg, Curtea de Arges, Romania. 50

Enguix, G. B. and Jiménez López, M. D. (2006). Linguistic Membrane Systems and Applications. In *Applications of Membrane Computing*, chapter 13, pages 347–388. Springer. 50

Fernández, L. (2006). New Algorithms for Application of Evolution Rules based on Applicability Benchmarks. In *International Conference on Bioinformatics and Computational Biology (BIOCOMP06)*, Las Vega, USA. 62, 63, 64, 79

Fernández, L., Martinez, V., Arroyo, F., and Mingo, L. (2005). A hardware circuit for selecting active rules in transition P systems. In *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, Timisoara, Romania. IEEE Computer Society. 62

Fondevilla, C., Colomer, M. A., Fillat, F., and Valencia-Cabrera, L. (2011). Plant communities dynamics model using P systems. In *Twelveth Conference on Membrane Computing*, pages 497–508, Fontainebleau, France. 48

Freund, R. and Verlan, S. (2007). A formal framework for static (tissue) p systems. In Eleftherakis, G., Kefalas, P., Păun, G., Rozenberg, G., and Salomaa,

A., editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 271–284. Springer Berlin Heidelberg. 37, 40, 42, 71, 80, 87, 96

Frisco, P. (2008). *Computing with Cells. Advances in Membrane Computing*. Oxford University Press. 36

García-Quismondo, M., Gutiérrez-Escudero, R., Pérez-Hurtado, I., Pérez-Jiménez, M. J., and Riscos-Núñez, A. (2010). An Overview of P-Lingua 2 . 0. In Păun, G., Pérez-Jiménez, M. J., Riscos-Núñez, A., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 264–288. Springer Berlin / Heidelberg. 54

Gheorghe, M., Ipate, F., and Dragomir, C. (2012). A kernel p system. *Tenth Brainstorming Week on Membrane Computing*, II:153–170. 37

Gramatovici, R. and Enguix, G. B. (2006). Parsing with P Automata. In *Applications of Membrane Computing*, chapter 14, pages 389–410. Springer. 50

Group, O. M. (2004). Meta object facility (mof) 2.0 core specification. 139

Group, O. M. (2015). Xml metadata interchange (xmi) specification. 139

Guerrero, G. D., Cecilia, J. M., García, J. M., Martínez-del Amor, M. A., Pérez-Hurtado, I., and Pérez-Jiménez, M. J. (2009). Analysis of P systems simulation on CUDA. In de Publicacions, U. d. C. n. S., editor, *Actas de las XX Jornadas de Paralelismo*, pages 219–224, La Coruña, Spain. 55

Gutiérrez, A., Fernández, L., Arroyo, F., and Alonso, S. (2008). Hardware and Software Architecture for Implementing Membrane Systems: A Case of Study to Transition P Systems. In Garzon, M. H. and Yan, H., editors, *DNA Computing*, volume 4848 of *Lecture Notes in Computer Science*, pages 211–220. Springer Berlin / Heidelberg. 64

Gutiérrez, A., Fernandez, L., Arroyo, F., and Martinez, V. (2006). Design of a Hardware Architecture Based on Microcontrollers for the Implementation of Membrane Systems. In *Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific ComputingEighth International Symposium on*

Symbolic and Numeric Algorithms for Scientific Computing, pages 350–353, Timisoara, Romania. Ieee. 64

Gutiérrez-Naranjo, M. A. and Pérez-Jiménez, M. J. (2011). Depth-first search with P systems. In Gheorghe, M., Hinze, T., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 257–264. Springer Berlin / Heidelberg. 50

Gutiérrez-Naranjo, M. A., Pérez-Jiménez, M. J., and Riscos-Núñez, A. (2005a). A Simulator for Confluent P Systems. In *Third Brainstorming Week on Membrane Computing*, pages 169–184, Seville, Spain. 53

Gutiérrez-Naranjo, M. A., Pérez-Jiménez, M. J., and Romero-Campero, F. J. (2005b). Simulating Avascular Tumors with Membrane Systems. In *Third Brainstorming Week on Membrane Computing*, pages 185–195, Seville, Spain. 48, 49

Gödel, K. (1931). Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik und Physik*, 38(1):173–198. 17

Hilbert, D. (1918). Axiomatisches Denken. *Mathematische Annalen*, 78(15):405. 17

Kefalas, P. and Stamatopoulou, I. (2011). Towards Modelling of Reactive, Goal-Oriented and Hybrid Intelligent Agents Using P Systems. In Gheorghe, M., Hinze, T., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 265–272. Springer Berlin / Heidelberg. 50

Kelemen, J., Kelemenova, A., and Gheorghe Păun (2004). On the power of a biochemically inspired simple computing model: P colonies. In *Workshop on Artificial Chemistry (ALIFE09)*, pages 82–86, Nashville, USA. 36

Kolovos, D., Rose, L., García-Domínguez, A., and Richard, P. (2015). *The Epsilon Book*. 140

Leporati, A. and Ferretti, C. (2010a). Modeling and Analysis of Firewalls by (Tissue-like) P Systems. In *Eighth Brainstorming Week on Membrane Computing*, pages 177–188, Seville, Spain. 50

Leporati, A. and Ferretti, C. (2010b). Modeling and Analysis of Firewalls by (Tissue-like) P Systems. *ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY*, 13(2):169–180. 50

Leporati, A., Zandron, C., Ferretti, C., and Mauri, G. (2007). Solving Numerical NP-Complete Problems with Spiking Neural P Systems. In Eleftherakis, G., Kefalas, P., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 336–352. Springer Berlin / Heidelberg. 49

Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55. 20

Macías-Ramos, L. F., Pérez-Hurtado, I., García-Quismondo, M., Valencia-Cabrera, L., Pérez-Jiménez, M. J., and Riscos-Núñez, A. (2011). A P – Lingua based Simulator for Spiking Neural P Systems. In *Twelveth Conference on Membrane Computing*, Fontainebleau, France. 54

Maliţa, M. (2000). Membrane Computing in Prolog. In *Workshop on Multiset Processing*, pages 159–175, Curtea de Argeş, Romania. 52

Manca, V. (2008). The metabolic algorithm for P systems: Principles and applications. *Theoretical Computer Science*, 404(1-2):142–155. 49

Manca, V. (2010). Fundamentals of Metabolic P Systems. In Păun, G., Rozenberg, G., and Salomaa, A., editors, *The Oxford Handbook of Membrane Computing*, chapter 19, pages 475–498. Oxford University Press. 49

Manca, V., Pagliarini, R., and Zorzan, S. (2009). Toward an MP Model of Non-Photochemical Quenching NPQ Phenomenon : A Biological Description. In Corne, D. W., Frisco, P., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 299–310. Springer Berlin / Heidelberg. 49

Marchetti, L. and Manca, V. (2011). A methodology based on MP theory for gene expression analysis. In *Twelveth Conference on Membrane Computing*, Fontainebleau, France. 49

Martinez, V., Arroyo, F., Gutiérrez, A., and Fernandez, L. (2006). Hardware Implementation of a Bounded Algorithm for Application of Rules in a Transition P-System. In *Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'06)*, pages 343–349, Timisoara, Romania. IEEE Computer Society. 63

Martinez, V., Fernandez, L., Arroyo, F., and Gutiérrez, A. (2007). HW Implementation of a Optimized Algorithm for the Application of Active Rules in a Transition P-System. *Information Technologies and Knowledge*, 14:324–331. 63

Martínez-del Amor, M. A., Pérez-Hurtado, I., García-Quismondo, M., Macías-Ramos, L., Valencia-Cabrera, L., Romero-Jiménez, A., Graciani, C., Riscos-Núñez, A., Colomer, M., and Pérez-Jiménez, M. (2013). Dcba: Simulating population dynamics p systems with proportional object distribution. In Csuhaj-Varjú, E., Gheorghe, M., Rozenberg, G., Salomaa, A., and Vaszil, G., editors, *Membrane Computing*, volume 7762 of *Lecture Notes in Computer Science*, pages 257–276. Springer Berlin Heidelberg. 56, 79

Martínez-del Amor, M. A., Pérez-Hurtado, I., Pérez-Jiménez, M. J., Cecilia, J. M., Guerrero, G. D., and García, J. M. (2009). Simulation of Recognizer P Systems by Using Manycore GPUs. In *Seventh Brainstorming Week on Membrane Computing*, pages 45–57, Seville, Spain. 55

Martínez-del Amor, M. A., Pérez-Hurtado, I., Pérez-Jiménez, M. J., Riscos-Núñez, A., and Colomer, M. A. (2010). A new simulation algorithm for multienvironment probabilistic P systems. In *IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*, pages 59–68, Liverpool, The UK. 54, 55, 65, 79

Martínez-del Amor, M. A., Pérez-Hurtado, I., Pérez-Jiménez, M. J., Riscos-Núñez, A., and Sancho-Caparrini, F. (2011). A simulation algorithm for multienviron-

ment probabilistic p systems: A formal verification. *International Journal of Foundations of Computer Science*, 22(01):107–118. 55, 79

Mauri, G., Leporati, A., and Zandron, C. (2009). Energy-Based Models of P Systems. In *Proceedings of the Tenth Workshop on Membrane Computing*, pages 104–124, Curtea de Arges, Romania. 36

Miller, J. and Mukerji, J. (2003). Mda guide v1.0.1. Technical report, Object Management Group. 138

Muskulus, M. (2009). Applications of Page Ranking in P Systems. In Corne, D. W., Frisco, P., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 311–324. Springer Berlin / Heidelberg. 50

Nepomuceno, I., Nepomuceno, J. A., and Romero-Campero, F. J. (2005). A Tool for Using the SBML Format to Represent P Systems which Model Biological Reaction Networks. In *Third Brainstorming Week on Membrane Computing*, pages 219–228, Seville, Spain. 53

Nepomuceno-Chamorro, I. A. (2004). A Java Simulator for Basic Transition P Systems. *Journal of Universal Computer Science*, 10(5):620–629. 53

Neumann, J. V. (1945). First draft of a report on the EDVAC. Technical report, University of Pennsylvania. 17

Nguyen, V. (2010). *An implementation of the parallelism, distribution and non-determinism of membrane computing models on reconfigurable hardware*. PhD thesis, Adelaide. 25, 65, 66, 69, 96

Nguyen, V., Kearney, D., and Gioiosa, G. (2008). An implementation of membrane computing using reconfigurable hardware. *COMPUTING AND INFORMATICS*, 27:551–569. Times Cited: 3. 80

Nguyen, V., Kearney, D., and Gioiosa, G. (2009). An algorithm for non-deterministic object distribution in p systems and its implementation in hardware. *MEMBRANE COMPUTING*, 5391:325–354. Times Cited: 4. 79

Nguyen, V., Kearney, D., and Gioiosa, G. (2010). A region-oriented hardware implementation for membrane computing applications. In Păun, G., Pérez-Jiménez, M., Riscos-Núñez, A., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 385–409. Springer Berlin Heidelberg. 80

Nishida, T. Y. (2006a). A Membrane Computing Model of Photosynthesis. In *Applications of Membrane Computing*, chapter 6, pages 181–202. Springer. 48

Nishida, T. Y. (2006b). Chapter 11 Membrane Algorithms : Approximate Algorithms for NP-Complete Optimization Problems. In Ciobanu, G., Păun, G., and Pérez-Jiménez, M. J., editors, *Applications of Membrane Computing*, chapter 11, pages 303–314. Springer. 49

NVIDIA (2010). *NVIDIA Cuda C Programming Guide Versión 3.2.* NVIDIA Corporation. 20

NVIDIA (2011). TESLA C2050 / C2070 GPU ComputinG Processor. 20

Pérez-Hurtado, I., Valencia-Cabrera, L., Pérez-Jiménez, M. J., Colomer, M. A., and Riscos-Núñez, A. (2010). MeCoSim: A general purpose software tool for simulating biological phenomena by means of P systems. In *IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*, pages 637–643, Changsha, China. IEEE. 54

Pérez-Jiménez, M. (2005). An approach to computational complexity in membrane computing. In Mauri, G., Păun, G., Pérez-Jiménez, M. J., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 3365 of *Lecture Notes in Computer Science*, pages 85–109. Springer Berlin / Heidelberg. 49

Pérez-Jiménez, M. and Romero-Campero, F. (2004). A CLIPS simulator for recognizer P systems with active membranes. In *Second Brainstorming Week on Membrane Computing*, pages 387–413, Seville, Spain. 53

Pérez-Jiménez, M. J. and Romero-Campero, F. J. (2007). Modelling EGFR signalling network using continuous membrane systems. In *Computational Methods*

in Systems Biology, International Conference, CMSB, volume i, Edinburgh, The U.K. 48

Pérez-Jiménez, M. J., Romero-Jiménez, A., and Sancho-Caparrini, F. (2006). Computationally Hard Problems Addressed Through P Systems. In Ciobanu, G., Păun, G., and Pérez-Jiménez, M. J., editors, *Applications of Membrane Computing*, chapter 12, pages 315–346. Springer. 49

Petreska, B. and Teuscher, C. (2004). A Reconfigurable Hardware Membrane System. In Martín-Vide, C., Mauri, G., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 269–285. Springer Berlin / Heidelberg, Tarragona, Spain. 25, 58, 59, 60, 80, 96

Păun, A. and Păun, G. (2002). The power of communication: P systems with symport/antiport. *New Generation Computing*, 20:295–305. 35

Păun, G. (2000). Computing with Membranes. *Journal of Computer and System Sciences*, 61:108–143. 18, 32, 35, 37, 43, 71, 96

Păun, G. (2001). P systems with active membranes: Attacking NP-Complete problems. *Journal of Automata, Languages and Combinatorics*, 6(May):75–90. 49

Păun, G. (2002). *Membrane Computing: An Introduction.* Springer. 35, 37

Păun, G., Rozenberg, G., and Salomaa, A., editors (2010). *The Oxford Handbook of Membrane Computing.* Oxford University Press. 35, 36, 37, 43

Păun, G. (2008). Bibliography of spiking neural P systems. *Natural Computing*, 7(4):551–553. 36

Rozenberg, G., Bäck, T., and Kok, J. N., editors (2012). *Handbook of Natural Computing.* Springer. 18

Rozenberg, G. and Salomaa, A., editors (1997). *Handbook of Formal Languages.* Springer-Verlag New York, Inc., New York, NY, USA. 81

Sanders, J. and Kandrot, E. (2011). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley. 20

Schmidt, D. (2006). Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31. 138

Siegel, J. (2015). Introduction to omg's unified modeling language® (uml®). Technical report, Object Management Group. 139

Stamatopoulou, I., Gheorghe, M., and Kefalas, P. (2005). Modelling dynamic organization of biology-inspired multi-agent systems with communicating x-machines and population p systems. In Mauri, G., Păun, G., Pérez-Jiménez, M. J., Rozenberg, G., and Salomaa, A., editors, *Membrane Computing*, volume 3365 of *Lecture Notes in Computer Science*, pages 389–403. Springer Berlin / Heidelberg. 50

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional. 138, 139

Suzuki, Y. and Tanaka, H. (2000). On a LISP Implementation of a Class of P Systems. *Romanian Journal of Information Science and Technology*, 3(2):173–186. 53

Syropoulos, A., Mamatas, E., Allilomes, P., and Sotiriades, K. (2004). A distributed simulation of transition P systems. *Membrane Computing*, pages 133–145. 56, 57

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265. 17

Weisstein, E. W. (2014). Padovan sequence. `http://mathworld.wolfram.com/PadovanSequence.html`. 86

Wilson, P. (2007). *Design recipes for FPGAs*. SciencieDirect. 22

Zeng, X., Adorna, H., and Angel, M. (2011). Matrix Representation of Spiking Neural P Systems. In Gheorghe, M., Hinze, T., Păun, G., Rozenberg, G., and

Salomaa, A., editors, *Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 377–391. Springer Berlin / Heidelberg. 56