Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Tesis Doctoral

Simulación de Sistemas Híbridos por Eventos Discretos: Tiempo Real y Paralelismo.

Lic. Federico Martín Bergero

Director: Dr. Ernesto Kofman

Miembros del Jurado: Dr. Maximiliano Cristiá
Dr. Pablo Lotito
Dr. Esteban Mocskos

Tesis presentada en la Facultad de Ciencias Exactas, Ingeniería y Agrimensura, en cumplimiento parcial de los requisitos para optar al título de

Doctor en Informática

Diciembre 2012

Certifico que el trabajo incluído en esta tesis es el resultado de tareas de investigación originales y que no ha sido presentado para optar a un título de posgrado en ninguna otra Universidad o Institución.

Lic. Federico Bergero

Resumen

Esta Tesis presenta nuevos resultados relacionados con la simulación de sistemas continuos mediante métodos de cuantificación de estados (QSS), resolviendo principalmente los problemas de utilización de estos algoritmos en tiempo real y en forma paralela.

Los métodos de QSS, que muestran gran eficiencia para simular sistemas con discontinuidades, aproximan los sistemas continuos mediante sistemas de eventos discretos bajo el formalismo DEVS.

Como punto de partida, se desarrolló entonces un algoritmo de simulación de DEVS en tiempo real y se implementó en la herramienta de simulación PowerDEVS. Esta implementación, bajo el sistema operativo de tiempo real RTAI–Linux, junto a las librerías desarrolladas de acceso al hardware, aporta de por sí un nuevo entorno de simulación y control basado en QSS para aplicaciones de tipo *Hardware in the Loop* y *Man in the Loop*.

A partir de este resultado, se desarrollaron dos nuevas técnicas de simulación en paralelo para métodos de QSS en arquitecturas multi–core que constituyen el resultado más importante de la Tesis. Estas nuevas técnicas se basan en partir el sistema y simular cada subsistema en un procesador distinto, sincronizando cada simulación contra una fracción del tiempo real. De esta manera, se consigue una sincronización implícita entre los procesos evitando la coordinación entre los distintos procesadores que constituían el principal obstáculo para explotar eficientemente el paralelismo en los métodos de QSS.

Estas técnicas, denominadas SRTS (*Scaled Real Time Synchronization*) y ASRTS (*Adaptive SRTS*), fueron implementadas en PowerDEVS en base al entorno de tiempo real ya mencionado y se realizaron estudios para determinar cómo escalan ante distintos números de procesadores. Asimismo, se realizó un análisis teórico del error numérico introducido por los retardos de comunicación intrínsecos a la implementación de las mismas.

Dado que la utilización del paralelismo está ligada a la necesidad de simular modelos de gran escala, se desarrolló también una extensión del formalismo DEVS para describir de manera sencilla modelos grandes que presentan conjuntos de componentes más simples replicados. Esta extensión, llamada Vectorial DEVS, fue también implementada en PowerDEVS y utilizada en la construcción de los ejemplos de prueba de los algoritmos de SRTS y ASRTS.

Finalmente, y con el objetivo de poder utilizar las técnicas aquí propuestas en un contexto más amplio de modelos y no limitados a sistemas descriptos en PowerDEVS, se desarrollaron e implementaron

dos metodologías que permiten simular con QSS modelos descriptos en el lenguaje estándar Modelica. La primer metodología consiste en convertir directamente un modelo de Modelica en un modelo de la aproximación QSS que produce PowerDEVS, de manera que sea esta última herramienta la que ejecuta la simulación. La segunda metodología consiste en convertir el modelo original en un sistema expresado en un subconjunto de Modelica que puede luego ser simulado por una herramienta de simulación autónoma de QSS.

Abstract

This Thesis presents new results related to continuous system simulation using quantized state methods (QSS), solving the main issues concerning real–time and parallel implementation of these algorithms.

QSS methods exhibit important advantages in the simulation of systems with frequent discontinuities and approximate continuous systems by discrete event systems within the DEVS formalism.

First, a real–time DEVS simulation algorithm was developed and implemented in the PowerDEVS simulation tool. This implementation, that executes under the Real Time Operating System RTAI–Linux, together with the hardware access layer library represent themselves a new environment for QSS–based simulation and control in *Hardware in the Loop* and *Man in the Loop* applications.

Making use of this result, two new techniques for QSS parallel simulation on multicore architectures were developed. These techniques, which constitute the main result of this Thesis, are based on splitting the system and simulating each sub–system on a different processor, synchronizing each process against a fraction of the physical wall time. In this way, an implicit synchronization is achived avoiding the costs of inter–processor coordination that constituted the main obstacle to exploit parallelism in QSS methods.

These techniques, called SRTS (*Scaled Real Time Synchronization*) and ASRTS (*Adaptive SRTS*), were implemented in PowerDEVS based on the real–time environment mentioned above. An extensive analysis on the speed-up against the number of processors and a theoretical study about the numerical errors introduced by communication delays were made.

As the usage of parallelization is closely related to the need of simulating large scale models, an extension to the DEVS formalism was developed in order to describe large models in a simple way. This extension called Vectorial DEVS, was also implemented in PowerDEVS and used in the construction of test cases for the SRTS and ASRTS algorithms

Finally, and with the goal of applying the proposed techniques in a wider context not limited to PowerDEVS-described models, two methodologies were developed and implemented for QSS simulation of systems described in the standard Modelica language. The first methodology automatically converts Modelica models into QSS–based DEVS models that are then simulated in PowerDEVS. The second methodology, converts the original Modelica model into a subset of Modelica in order to be simulated with a stand–alone QSS simulation tool.

Reconocimientos

Esta Tesis fue realizada con la financiación del Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) mediante becas tipo I y tipo II durante los años 2008 a 2012 y con subsidios de los proyectos OPENPROD-ITEA2 *CTI grant Nr.12101.1;3 PFES-ES* y CONICET *PIP 00183*.

Agradecimientos

Este trabajo no hubiera sido posible sin el aporte de muchas personas.

Entre ellas, quiero agradecer a mi director, Ernesto Kofman que ha motivado en mi la búsqueda de conocimiento y ha soportado dudas, consultas y preguntas. Agradecer también a mis compañeros durante este viaje, Gustavo Migoni, Fernando Fontenla, Mario Bortolotto, Matías Nacuse, Joaquín Fernández y también a mis compañeros de antaño (Rafael Namías, Diego Hollmann, Iván Ramello, Mariano Salvetti y Eric Biagioli) junto con toda la gente del CIFASIS, el Departamento de Computación y el de Posgrado de la FCEIA.

Por otro lado quiero agradecer a François Cellier, Xenofontas Floros, Gabriel Wainer y Mario Munich que me han dado la oportunidad de trabajar con ellos conociendo así distintas partes del mundo y de la ciencia.

Un agradecimiento especial para los jurados de esta Tesis, Maximiliano Cristiá, Pablo Lotito y Esteban Mocskos, que con sus comentarios y correcciones elevaron la calidad de este trabajo.

En lo personal quisiera agradecer a mis padres Rubén y Beatriz, a mi hermana Ana Laura, a mi novia Ivana y amigos (Gastón, Ezequiel, Nicolás, Poroto y Fede), que fueron los pilares emocional para poder completar esta etapa de mi vida.

A todos Uds, gracias de corazón.

Fede

Índice general

Lista de Acrónimos 1

1. Introducción 3
1.1. Organización de la Tesis 5
1.2. Aportes Originales 5
1.3. Trabajo Relacionado y Relevancia de los Resultados 6
1.4. Publicaciones de Apoyo 8

2. Conceptos Previos 9
2.1. Métodos Clásicos de Integración Numérica 9
2.1.1. Métodos de Euler 10
2.1.2. Precisión de las Aproximaciones 11
2.1.3. Estabilidad Numérica de un Método de Integración 11
2.1.4. Métodos Monopaso 13
2.1.5. Algoritmos de Control de Pasos 14
2.1.6. Métodos Multipaso 14
2.1.7. Sistemas con Discontinuidades 15
2.1.8. Sistemas Rígidos (Stiff) 16
2.2. Quantized State Systems (QSS) 16
Métodos QSS de Alto Orden 18
2.2.1. Propiedades Teóricas de los métodos de QSS 19
2.2.2. Cuantificación Logarítmica 20
2.3. Discrete EVent Systems (DEVS) 20
2.3.1. Modelos DEVS Atómicos 21
2.3.2. Modelos DEVS Acoplados 22
2.3.3. Simulación de Modelos DEVS 24
2.3.4. Herramientas de Simulación DEVS 25
2.4. Formalismo DEVS y QSS 26
2.5. Herramienta de Simulación PowerDEVS 27
2.6. Sistemas de Tiempo Real 28
2.6.1. Sistemas Operativos de Tiempo Real (RTOS) 28
2.6.2. RealTime Application Interface – RTAI 29
2.7. Simulación en Paralelo de Eventos Discretos 30
2.7.1. Plataformas de Paralelización 31
2.7.2. Algoritmos de Sincronización 31
Sincronización CMB 31
Sincronización Optimista – TimeWarp 32
NoTime 33

3. Simulación DEVS con PowerDEVS en Tiempo Real **35**
3.1. Motor de Simulación de PowerDEVS 35
3.2. Implementación de PowerDEVS–RTAI 37
3.2.1. Algoritmo de Simulación – Sincronización y Tiempo Real . 37
3.2.2. Manejo de Interrupciones y Archivos 38
3.2.3. Medición del Tiempo 39
3.2.4. Desempeño en Tiempo–Real 39
3.2.5. Latencia de Interrupción 40
3.3. Biblioteca de Modelos para Tiempo–Real 41
3.4. Comunicación con Scilab 42
El Lado de Scilab 42
El Lado de PowerDEVS 43
Parámetros desde Scilab y Bloques de Interfaz 44
3.5. Ejemplos y Resultados 45
3.5.1. Ripple vs Frecuencia en una Fuente Conmutada Buck . . . 45
3.5.2. Control de un Motor en RT 46
3.6. Conclusiones 50

4. Modelos DEVS Vectoriales **51**
4.1. Introducción 51
4.2. Otros Formalismos y Herramientas 52
4.3. Modelos Vectorial DEVS 54
4.3.1. Ejemplo Motivador 54
4.3.2. Definición de Modelos DEVS Vectoriales 55
4.3.3. Definición de Vectorial DEVS 56
4.3.4. Comportamiento de Modelos Vectorial DEVS 56
4.3.5. Modelos de Interface para Vectorial DEVS 58
4.3.6. Conectando Vectorial DEVS con Modelos DEVS Clásicos . 59
4.4. Implementación PowerDEVS de Modelos Vectorial DEVS 60
4.4.1. Implementación de los Eventos 60
4.4.2. Implementación de los Modelos Vectorial DEVS 61
4.4.3. Inicialización del los Modelos Vectorial DEVS 63
4.4.4. Creando un Modelo Vectorial DEVS 63
4.5. Particionado Automático de modelos Vectorial DEVS 64
4.5.1. Algoritmo de Particionado 64
Particionado de Modelos Puramente Vectorial DEVS 65
4.5.2. Particionado en Presencia de Modelos DEVS de Interface . 65
4.5.3. Algoritmo de Particionado de Modelos Vectorial DEVS . . 68
4.6. Ejemplos y Resultados 68
4.6.1. Línea de Transmisión LC 68
4.6.2. Control de Energía de un Conjunto de Aires Acondicionados 69
4.6.3. Redes Neuronales Pulsantes 72
4.6.4. Modelo en Espacio de Estados 76
4.7. Conclusiones 77

5. Simulación en Paralelo - SRTS y ASRTS **79**
5.1. Scaled Real Time Synchronization – SRTS 80
5.1.1. Idea Básica 80
5.1.2. Algoritmo Scaled Real Time Synchronization 82
5.1.3. SRTS y Error Numérico en los Métodos de QSS 85
5.1.4. Implementación en PowerDEVS– RTAI 85
5.2. Adaptive Scaled Real Time Syncrhonization – ASRTS 87
5.2.1. Idea Básica 87
5.2.2. Algoritmo de Adaptive SRTS 88
5.2.3. Factor Óptimo de Escalado 89
5.2.4. Selección de Parámetros 89
5.3. Relación con otros algoritmos 90
5.4. Ejemplos y Resultados 90
5.4.1. Control de Energía de un Conjunto de Aires Acondicionados 91
5.4.2. Cadena de Inversores Lógicos 93
5.4.3. Línea de Transmisión LC 96
5.5. Conclusiones 100

6. Simulación de Modelos Modelica **101**
6.1. Simulación de modelos Modelica en PowerDEVS 101
6.1.1. Simulación de Modelica con QSS 102
Describiendo Discontinuidades en Modelica 102
Estructura DEVS 103
6.1.2. Interfaz OpenModelica – PowerDEVS (OMPD) 105
¿Qué Necesita PowerDEVS ? 105
¿Qué provee OpenModelica? 106
6.1.3. Resultados 107
Rectificador de Media Onda 109
Fuente Conmutada 111
6.2. Simulación Autónoma de Modelos Modelica 113
6.2.1. Simulador Autónomo de QSS 114
6.2.2. Modelica y el Simulador Autónomo de QSS 115
6.2.3. El Subconjunto μ–Modelica 116
6.2.4. Simulando μ–Modelica con el Simulador Autónomo 116
6.2.5. Conversión de Modelica a μ–Modelica 117
6.2.6. Ejemplos y Resultados 119
Fuente Conmutada 120
Circuito Interleaved DC-DC 123
6.3. Conclusiones 128

7. Conclusiones Generales **129**
7.1. Conclusiones Generales 129
7.2. Trabajo a Futuro y Problemas Abiertos 130

Bibliografía **133**

Lista de Acrónimos

ASRTS Adaptive Scaled Real Time Synchronization

CMB Chandy Misra Bryant

CUDA Compute Unified Device Architecture

DAE Differential Algebraic Equations

DEVS Discrete EVent System

EIC External Input Connection

EOC External Output Connection

GPGPU General-Purpose computing on Graphics Processing Units

GUI Graphical User Interface

HIL Hardware In the Loop

IPC Inter Process Communication

LTW Light Time Warp

MIL Man In the Loop

MIMD Multiple Instructions and Multiple Data

MPI Message Passing Interface

ODE Ordinary Differential Equation

OMPD OpenModelica PowerDEVS

PDES Parallel Discrete Events Simulation

PWM Pulse Width Modulation

QSS Quantized State Systems

RTAI Real Time Application Interface

RTDEVS Real Time Discrete EVent System

RTOS Real Time Operating System

SRTS Scaled Real Time Synchronization

Capítulo 1

Introducción

La simulación por computadoras se ha convertido en una herramienta fundamental a la hora de estudiar, desarrollar y analizar modelos matemáticos de diversos ámbitos (físicos, económicos, biológicos).

Estos modelos se expresan como sistemas continuos representados generalmente mediante ecuaciones diferenciales. En presencia de discontinuidades y/o sub–sistemas discretos se denominan sistemas híbridos. Debido a que gran parte de las ecuaciones diferenciales no poseen solución analítica, éstas deben ser discretizadas para luego simularlas mediante *métodos de integración numérica*. La mayoría de los métodos de integración numérica discretizan la variable independiente (generalmente el tiempo) transformando el sistema en un conjunto de ecuaciones en diferencias [21].

Se han desarrollado recientemente diversos métodos que difieren del enfoque clásico ya que en vez de discretizar la variable independiente se basan en la discretización de los estados. Esta formulación, llamada QSS [21], transforma el sistema continuo original en un sistema de eventos discretos que puede ser representado por el formalismo Discrete EVent System Specification (DEVS) [94]. Al ser representado mediante eventos discretos, los métodos de QSS pueden tratar con discontinuidades mucho más eficientemente que los métodos clásicos.

DEVS es el formalismo más general para describir sistemas de eventos discretos. La simulación de modelos DEVS es muy sencilla y se han desarrollado distintas herramientas de modelado y simulación DEVS. Entre ellas encontramos a PowerDEVS [11], un simulador DEVS de propósito general que incluye también una implementación completa de la familia de métodos de QSS.

Muchas aplicaciones requieren que las simulaciones se realicen en *Tiempo–Real*, es decir, que el tiempo de simulación se mantenga lo más próximo posible al tiempo físico [21]. Ejemplos de estas aplicaciones incluyen a los sistemas *Hardware In the Loop (HIL)* y *Man In the Loop (MIL)*.

Las restricciones impuestas por este tipo de simulación abren una serie de problemas de relativa complejidad que involucran cuestiones de sistemas operativos, entrada–salida de datos, interrupciones, etc. Por esto, el desarrollo de una herramienta de simulación de propósito general de Tiempo–Real es una tarea compleja, pero que a su vez tiene una aplicabilidad muy importante y amplia.

Los métodos de QSS poseen características que los hacen aptos para su simulación en tiempo real. Primero, evitan la iteración para la detección de discontinui-

dades presente en los métodos clásicos. Segundo, hay métodos dentro de la familia, como los LIQSS(Linearly Implicit QSS), que son aptos para sistemas *stiff* y son explícitos, es decir, no iteran en cada paso de simulación. Estas dos características hacen que los métodos tengan un costo computacional acotable, un requisito fundamental a la hora de simularlos en tiempo real.

Previo a esta Tesis, no se había investigado la implementación de éstos métodos en tiempo real. Una primera implementación incompleta fue realizada por el autor en [9]. En este trabajo extendimos a PowerDEVS con la capacidad de realizar simulaciones en Tiempo Real y un conjunto de rutinas de acceso a hardware de bajo nivel, obteniendo así un marco para el desarrollo de experimentos MIL y HIL. Este entorno también fue utilizado para desarrollar métodos de simulación en paralelo como veremos luego.

Aunque el algoritmo de simulación de sistemas de eventos discretos es sencillo, la simulación en sí misma puede volverse muy costosa debido al tamaño del modelo o a la complejidad de computar cada paso. Para ello existen técnicas de simulación en paralelo llamadas *Parallel Discrete Event Simulation (PDES)* que se basan en particionar el modelo en sub–sistemas llamados *procesos físicos* y simular cada uno de ellos en distintos Procesadores Lógicos. Mientras que PDES reduce el costo computacional, introduce un nuevo problema relacionado con la necesidad de sincronización entre procesadores. Muchos trabajos han propuesto soluciones a este problema [19, 47, 65, 79] de distintas formas.

Antes de este trabajo la simulación en paralelo de los métodos de QSS habían sido vagamente investigada. Dos antecedentes fueron realizados sobre procesadores GPU [60, 84] donde no se obtuvieron buenos resultados concluyendo que los GPU no son aptos para los métodos de QSS.

En esta Tesis presentamos también *dos nuevas técnicas* para la simulación en paralelo de sistemas continuos en DEVS, ambas basadas en la sincronización de los Procesadores Lógicos con una versión escalada del tiempo físico. En la primera, denominada Scaled Real–Time Synchronization (SRTS), el factor de escalado es un parámetro del algoritmo que queda a elección del usuario. Desarrollamos luego una versión adaptiva del algoritmo llamada Adaptive–SRTS (ASRTS) en la cual este factor se ajusta dinámicamente dependiendo de la carga del sistema.

Ambas técnicas fueron implementadas en PowerDEVS validando su funcionamiento. Como necesitábamos una sincronización temporal precisa entre los Procesadores Lógicos utilizamos la versión antes mencionada de PowerDEVS que se ejecuta en un sistema operativo de tiempo real llamado, RealTime Application Interface (RTAI) [26].

Naturalmente uno utilizaría estas técnicas en simulaciones de sistemas de gran escala o muy complejos ya que de no ser así, no valdría la pena aplicar estos algoritmos.

Muchos formalismos permiten describir modelos de forma gráfica, como por ejemplo Redes de Petri, o el mismo DEVS. En cuanto el tamaño del modelo empieza a crecer, su representación gráfica se torna más y más difícil de comprender la estructura subyacente, y se vuelve casi imposible trabajar con él. Otro problema surge cuando el modelo tiene una estructura de conexión compleja o demasiado grande, en donde el modelador tiene que dibujar una por una las conexiones entre sub–sistemas. En estos casos uno se ve obligado a dejar de lado las representaciones gráficas y utilizar alguna notación escrita o textual donde se puedan definir grandes modelos sin escribir una descripción igual de grande. Un ejemplo de notación

escrita es el lenguaje Modelica [32, 35], que permite replicar y conectar modelos de forma programática. La mayoría de las herramientas de simulación de Modelica utilizan métodos de integración clásicos (Runge-Kutta o DASSL) los cuales presentan problemas al integrar sistemas híbridos. Previamente al presente trabajo no se había investigado la utilización de los métodos de QSS fuera de DEVS. Esto se debe a que QSS necesita cierta información estructural que no está explícita en una representación como la de Modelica.

En este aspecto hemos hecho dos aportes. Primero hemos definido una extensión al formalismo DEVS, llamada Vectorial DEVS, para facilitar la definición de modelos grandes y la hemos implementado en PowerDEVS junto con diversas formas de definir convenientemente las conexiones.

Luego hemos desarrollado e implementado dos metodologías para la simulación de modelos descriptos en Modelica. En la primera el modelo descripto en Modelica es convertido automáticamente a un modelo DEVS y luego simulado en PowerDEVS mediante los métodos de QSS. En la segunda metodología, el modelo Modelica es procesado y convertido en un lenguaje más simple que luego es simulado por la herramienta *stand–alone* de simulación desarrollada recientemente por Joaquín Fernández [28] que implementa la totalidad de los algoritmos de QSS.

1.1. Organización de la Tesis

Este primer capítulo presenta un resumen del trabajo realizado junto con un breve análisis del estado del arte. En el capítulo 2 se describe algunos conceptos básicos necesarios utilizados en el resto de la Tesis.

Seguidamente en el capítulo 3 presentamos las extensiones realizadas a PowerDEVS para su ejecución en Tiempo Real mientras que en el capítulo 4 se describe una extensión al formalismo DEVS para describir modelos vectoriales.

En el capítulo 5 se describen las nuevas técnicas de simulación en paralelo que fueron implementadas con lo presentado en los capítulos anteriores. Luego en el capítulo 6 introducimos dos metodologías para la simulación de modelos Modelica a través de los métodos de QSS. En los cuatro capítulos se muestran ejemplos de aplicación y un estudio de su funcionamiento.

Finalmente en el capítulo 7 presentamos las conclusiones generales de la Tesis y delineamos algunos trabajos a futuro que pretendemos realizar.

1.2. Aportes Originales

A partir del capítulo 3 en adelante los resultados presentados son originales. Las principales contribuciones de esta Tesis se enmarcan dentro de tres categorías distintas. Primero se extendió el software PowerDEVS para su ejecución en tiempo real que luego fue utilizado cómo herramienta experimental de los siguientes desarrollos (capítulo 3).

Se introdujeron dos nuevas técnicas de simulación en paralelo SRTS y ASRTS que solucionan los problemas de las técnicas previamente existentes de una manera novedosa. Estas técnicas, aunque generales en su formulación, fueron adaptadas a la simulación de sistemas híbridos mediante los métodos de QSS (capítulo 5).

Con la motivación de describir modelos de gran escala de una forma adecuada, se introdujo una extensión al formalismo DEVS para modelado vectorial y

dos metodologías para la simulación de modelos descriptos en Modelica mediante métodos de QSS (capítulos 4 y 6 respectivamente).

1.3. Trabajo Relacionado y Relevancia de los Resultados

Existen distintos trabajos relacionados con esta Tesis. En el área de simulación en Tiempo Real existen varios antecedentes. El *Real Time Workshop* [50] es una herramienta de la empresa MathWorks incluida en Matlab. Permite describir sistemas mediante diagramas de bloques y luego generar código C++ que puede ejecutarse en diversos Real Time Operating Systems (RTOS).

Dentro del ámbito de DEVS existen también implementaciones de simuladores en tiempo real. Uno de ellos es CD++ [89] (basado en el formalismo Cell-DEVS). Hong [45] presenta también una extensión de DEVS para tiempo real llamada RTDEVS y su implementación en un simulador DEVS.

Nuestro aporte en este caso introduce PowerDEVS como una nueva herramienta de simulación DEVS en tiempo real (basada completamente en software libre) con una completa implementación de los métodos de integración QSS. Junto con la librería de acceso a hardware de bajo nivel, PowerDEVS se conforma en un entorno para experimentos MIL y HIL que puede ejecutarse en hardware de bajo costo (PC). En comparación con las herramientas existentes, este nuevo entorno permite la descripción de sistemas continuos (lo cual es más complejo en CD++) y posee la ventaja de utilizar los métodos de QSS los cuales no están implementados en ningún otra herramienta.

La simulación DEVS en paralelo ha sido estudiada desde principios de los 1980's produciendo distintos algoritmos de sincronización entre los Procesadores Lógicos. Uno de los primeros es el algoritmo propuesto por Chandy, Misra y Bryant (CMB) [19, 65] el cual implementa una sincronización conservadora, esto es, los procesadores no avanzan su tiempo de simulación hasta que sepan que no recibirán un evento con un tiempo menor.

Un segundo enfoque [47] propone una sincronización optimista relajando la restricción causal sobre los eventos, permitiendo que los procesadores avancen su tiempo de simulación tan rápido como puedan y realizando correcciones cuando se detecta una violación de la causalidad.

Por último, una técnica llamada NOTIME [79] investiga los efectos de no sincronizar ningún procesador sabiendo que la solución puede no ser correcta.

En la literatura existen algunos antecedentes de la utilización de PDES para la simulación de sistemas híbridos.

Tang et al. presentó en [86] un estudio de un algoritmo PDES optimista aplicado a ecuaciones diferenciales parciales donde se obtuvo una ganancia de performance lineal de hasta cuatro veces. Trabajos similares han reportado la aplicación de PDES a la simulación de sistemas físicos como el artículo de Bauer [6] (basado en el artículo de Nutaro [72]), concluyendo que los algoritmos optimistas de PDES no son aptos para estos problemas si no vienen acompañados de un balance de carga dinámico.

En el área de simulación en tiempo real, existen resultados acerca de la simulación distribuida de sistemas. Adelantado [1] presenta un middleware High Level Architecture (HLA) con capacidades de tiempo real junto con un modelo para su

validación. Como veremos, la técnica utilizada allí se relaciona con nuestro trabajo en el uso de un reloj global para la sincronización. En [1] se supone que cada Procesador Lógico conoce a priori que las tareas se ejecutarán con una frecuencia dada y que se conoce también el tiempo de ejecución en el peor caso. Estas suposiciones no se cumplen en nuestro escenario donde no existe una frecuencia de ejecución de cada tarea y el tiempo del peor caso puede variar durante la simulación.

Muchos trabajos tratan sobre el problema de simular modelos DEVS en paralelo utilizando las técnicas previamente mencionadas. Jafer y Wainer presentan en [46] CCD++, un simulador de DEVS y Cell–DEVS que utiliza un algoritmo conservador y realizan comparaciones con un algoritmo optimista. Kim et al. presentan en [49] una metodología para la simulación distribuida de modelos DEVS basada en TimeWarp. Un enfoque mixto combinando algoritmos optimistas y conservadores que permite explotar al máximo el *look–ahead* es estudiado por Praehofer [78].

Liu [58] explora una versión liviana del mecanismo de TimeWarp para simulación de modelos DEVS (y Cell–DEVS) implementado en un procesador Cell [76]. Allí introduce *Lightweight Time Warp* o LWT y *Multicore Accelerated DEVS Systems* como medios para subsanar los cuellos de botella de TimeWarp y para ocultar al usuario la complejidad de la programación en multicores.

Finalmente, Hong et al. [45] proponen una extensión al formalismo DEVS, llamada RealTime DEVS (o RTDEVS) para modelar sistemas de tiempo real. La idea detrás de RTDEVS es llenar los tiempos de avance con tareas ejecutables y especificar cotas temporales para cada actividad. También implementan un simulador DEVS que sincroniza el tiempo físico con el de simulación. Como será visto en el Capítulo 5.3 esta técnica tiene una relación con lo presentado en esta Tesis.

Algunos resultados previos también han sido obtenidos en la paralelización de los métodos de QSS aplicados a sistemas continuos. Nutaro [71] investigó una primera implementación de QSS1 en un simulador DEVS paralelo. Allí estudia el uso del enfoque TimeWarp y concluye que no es adecuado para la simulación de grandes sistemas continuos. Usando un método de primer orden de QSS, Nutaro realizó simulaciones sobre un problema de mecánica de fluidos llamado “sod shock tube“ alcanzado una aceleración de 3 veces usando 6 procesadores pero decrecía rápidamente al usar más procesadores.

Otra implementación de los métodos de QSS en una arquitectura multi procesador fue investigada por Magio et al. [61]. Esta implementación fue realizada usando CUDA (un lenguaje para GPGPU). Desafortunadamente, como fue mencionado en esta Tesis, las unidades de procesamiento gráficas (GPU) ofrecen un juego de instrucciones limitadas y la sincronización de tareas es muy costosa. Además, las GPUs siguen una taxonomía de Flynn de *Single Instruction, Multiple Data (SIMD)* donde todos los procesadores ejecutan el mismo código en paralelo, lo cual reduce considerablemente la performance cuando el código tiene ramas divergentes. Estos problemas limitan la clase de aplicaciones donde las GPUs puede ser usadas eficientemente. En el ejemplo analizado en el trabajo citado, se obtuvo una aceleración de hasta 7.2 veces para un sistema de 64 estados usando 64 procesadores lo cual es muy lejano a la aceleración óptima.

En el ámbito de modelado y simulación de grandes sistemas existen distintos formalismos. Uno de ellos es Modelica [32, 35], un lenguaje multi–dominio, orientado a objetos que permite describir modelos mediante sus ecuaciones fundamentales. El lenguaje está estandarizado por una asociación sin fines de lucro

que ha desarrollado también una extensa librería. Existen varios simuladores de código abierto como OpenModelica [33], JModelica [2] y también hay herramientas comerciales como Dymola [74].

Nuestro aporte en este caso permite simular modelos descriptos en Modelica utilizando los métodos de integración de QSS los cuales son adecuados para sistemas con discontinuidades frecuentes. Esto brinda al usuario de Modelica la posibilidad de simular una mayor cantidad de sistemas eficientemente.

Una forma de modelar grandes sistemas es definir su funcionamiento en base al de sus sub–componentes y sus interconexiones. Tanto Cell–DEVS [90] como StateCharts [41] están dentro de esta clase de formalismos, donde se puede dar una descripción de un sub–modelo y replicarlo en una forma matricial (en el caso de Cell–DEVS) o vectorial (en el caso de StateCharts). Ni StateCharts ni Cell–DEVS poseen una forma de representar sistemas continuos mediante QSS.

Siguiendo esta idea, proponemos una extensión al formalismo DEVS, llamada Vectorial DEVS que permite definir modelos vectoriales junto con diversas formas de conectarlos. De esta forma podemos representar grande sistemas continuos replicando componentes que los aproximen mediante QSS. Realizamos también una primera implementación de Vectorial DEVS en PowerDEVS demostrando su viabilidad y sus ventajas.

1.4. Publicaciones de Apoyo

Los resultados de esta Tesis fueron publicados en revistas y anales de conferencias. El primer resultado fue el desarrollo del motor de simulación de PowerDEVS junto con su ejecución sobre el sistema operativo de Tiempo Real RTAI. Estos resultados fueron publicados primero en conferencias locales [5, 12, 27] e internacionales [66], y luego en una revista internacional [11].

Posteriormente, las nuevas técnicas de paralelización SRTS y ASRTS destinadas a la simulación de sistemas de eventos discretos en una arquitectura multicore junto con varios casos de estudios y análisis del error introducido fueron publicados en un congreso local [14] y en una revista internacional [13].

Los resultados obtenidos en el área de modelado de sistemas a gran escala, y los desarrollos de metodologías multi-formalismo para Modelica fueron publicados en congresos internacionales [10, 29].

Hay también un artículo en desarrollo que contendrá las técnicas de modelado vectorial Vectorial DEVS y posibles formas de particionar un problema para su simulación en paralelo con las técnicas antes presentadas.

Capítulo 2

Conceptos Previos

En este capítulo se presenta una breve introducción de los conceptos necesarios para comprender el resto de la Tesis.

Primero se realiza una reseña de los enfoques clásicos utilizados para la simulación de sistemas híbridos (Sección 2.1) y en la Sección 2.2 el enfoque basado en los métodos de QSS.

Luego introducimos el formalismo DEVS (Sección 2.3), su relación con los métodos de QSS (Sección 2.4) y presentamos algunas herramientas de simulación, en particular PowerDEVS en la Sección 2.5.

Finalmente, son presentados los resúmenes de distintas técnicas de simulación en tiempo real y en paralelo en las secciones 2.6 y 2.7 respectivamente.

2.1. Métodos Clásicos de Integración Numérica

Un sistema continuo puede ser descripto por un modelo en espacios de estados de la forma:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \tag{2.1}$$

donde $\mathbf{x} \in \Re^n$ es el vector de estados, $\mathbf{u} \in \Re^m$ es una función de entradas conocidas, t representa el tiempo y con sus condiciones iniciales:

$$\mathbf{x}(t = t_0) = \mathbf{x}_0 \tag{2.2}$$

Sea $x_i(t)$ la trayectoria del estado i-ésimo expresada como función de tiempo simulado. Mientras que la ecuación 2.1 no contenga discontinuidades $x_i(t)$ será una función continua con derivada continua. Ésta puede ser aproximada con la precisión deseada mediante series de Taylor en cualquier punto de su trayectoria.

Denominando t^* al instante de tiempo en torno al cual se aproxima la trayectoria mediante una serie de Taylor, y siendo $t^* + h$ el instante de tiempo en el cual se quiere evaluar la aproximación, entonces, la trayectoria en dicho punto puede expresarse como sigue:

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

Reemplazando con la ecuación de estado (2.1), la serie (2.3) queda:

$$x_i(t^* + h) = x_i(t^*) + \mathbf{f_i}(t^*) \cdot h + \frac{df_i(t^*)}{dt} \cdot \frac{h^2}{2!} + \ldots \quad (2.4)$$

Los distintos algoritmos de integración difieren en la manera de aproximar las derivadas superiores de $\mathbf{f}$ y en el número de términos de la serie de Taylor que consideran para la aproximación.

2.1.1. Métodos de Euler

El algoritmo de integración más simple se obtiene truncando la serie de Taylor tras el término lineal:

$$\mathbf{x}(t^* + h) \approx \mathbf{x}(t^*) + \dot{\mathbf{x}}(t^*) \cdot h \quad (2.5a)$$

o:

$$\mathbf{x}(t^* + h) \approx \mathbf{x}(t^*) + \mathbf{f}(\mathbf{x}(t^*), t^*) \cdot h \quad (2.5b)$$

El parámetro h, que define la distancia entre dos instantes de tiempo donde calculamos la solución, se denomina *paso de integración*.

Este esquema es particularmente simple ya que no requiere aproximar ninguna derivada de orden superior, y el término lineal está directamente disponible del modelo de ecuaciones de estado. Este esquema de integración se denomina *Método de Forward Euler* (FE).

La Figura 2.1 muestra una interpretación gráfica de la aproximación realizada por el método de FE.

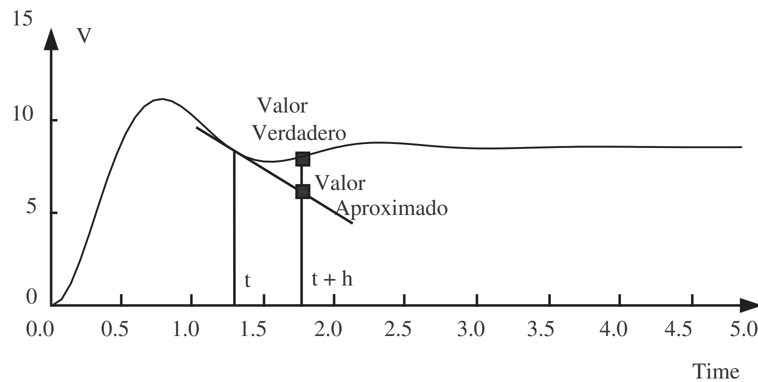

Figura 2.1: Integración numérica utilizando Forward Euler.

La simulación utilizando el método de FE se torna trivial ya que el método de integración utiliza sólo valores pasados de las variables de estado y sus derivadas. Un esquema de integración que exhibe esta característica se denomina *algoritmo de integración explícito*.

Otro método de integración numérica muy conocido, inspirado en el anterior y que recibe el nombre de *Método de Backward Euler*, reemplaza la fórmula de la

Ec.(2.5a) por la siguiente:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\ \mathbf{f}(\mathbf{x}_{k+1}, t_k) \tag{2.6}$$

El método de Backward Euler tiene un pequeño inconveniente: de acuerdo a la Ec.(2.6) para calcular $\mathbf{x}_{k+1}$ necesitamos conocer $\mathbf{x}_{k+1}$.

Naturalmente, conociendo $\mathbf{x}_k$ podemos resolver de alguna forma la Ec.(2.6) y obtener de allí el valor de $\mathbf{x}_{k+1}$. Por este motivo, se dice que el método de Backward Euler es un *Método Implícito*, ya que para encontrar cada valor debemos resolver una ecuación.

Cuando la función $\mathbf{f}$ es lineal, la resolución de la ecuación implícita (2.6) es muy sencilla (si bien requiere invertir una matriz). Por el contrario, en el caso no lineal, en general necesitaremos utilizar algún algoritmo iterativo que encuentre la solución para cada instante de tiempo. Normalmente, se utiliza la *iteración de Newton*.

2.1.2. Precisión de las Aproximaciones

Si el método coincide con la expansión en serie de Taylor hasta el término n-ésimo, se dice que el método es de orden n. Evidentemente, la precisión con la que se aproximan las derivadas de orden superior debe estar acorde al número de términos de la serie de Taylor que se considera. Si se tienen en cuenta $n+1$ términos de la serie, la precisión de la aproximación de la derivada segunda del estado $d^2x_i(t^*)/dt^2 = df_i(t^*)/dt$ debe ser de orden $n-2$, ya que este factor se multiplica por h^2. La precisión de la tercer derivada debe ser de orden $n-3$ ya que este factor se multiplica por h^3, etc. De esta forma, la aproximación será correcta hasta h^n. Luego, n se denomina *orden de la aproximación* del método de integración, o, simplemente, se dice que el método de integración es de orden n.

Mientras mayor es el orden de un método, más precisa es la estimación de $x_i(t^* + h)$. En consecuencia, al usar métodos de orden mayor, se puede integrar utilizando pasos grandes. Por otro lado, al usar pasos cada vez más chicos, los términos de orden superior de la serie de Taylor decrecen cada vez más rápido y la serie de Taylor puede truncarse antes.

El costo de cada paso depende fuertemente del orden del método en uso. En este sentido, los algoritmos de orden alto son mucho más costosos que los de orden bajo. Sin embargo, este costo puede compensarse por el hecho de poder utilizar un paso mucho mayor y entonces requerir un número mucho menor de pasos para completar la simulación. Esto implica que hay que buscar una solución de compromiso entre ambos factores.

2.1.3. Estabilidad Numérica de un Método de Integración

Consideremos el sistema lineal y autónomo:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \tag{2.7}$$

con condiciones iniciales como en la Ecuación 2.2. Su solución analítica es:

$$\mathbf{x}(t) = \exp(\mathbf{A} \cdot t) \cdot \mathbf{x_0} \tag{2.8}$$

la cual será *analíticamente estable* si todas las trayectorias permanecen acotadas cuando el tiempo tiende a infinito. El sistema (2.7) es analíticamente estable si y sólo si todos los autovalores de $\mathbf{A}$ tienen parte real negativa:

$$\mathbb{R}\text{e}\{\text{Eig}(\mathbf{A})\} = \mathbb{R}\text{e}\{\lambda\} < 0{,}0 \tag{2.9}$$

Aplicando entonces el algoritmo de FE a la solución numérica de este problema colocando el sistema de la Ec.(2.7) en el algoritmo de la Ec.(2.5a), se obtiene:

$$\mathbf{x}(t^* + h) = \mathbf{x}(t^*) + \mathbf{A} \cdot h \cdot \mathbf{x}(t^*) \tag{2.10}$$

que puede reescribirse en forma más compacta como:

$$\mathbf{x}(k + 1) = [\mathbf{I}^{(\mathbf{n})} + \mathbf{A} \cdot h] \cdot \mathbf{x}(k) \tag{2.11}$$

donde $\mathbf{I}^{(\mathbf{n})}$ es una matriz identidad de la misma dimensión que $\mathbf{A}$, es decir, $n \times n$. En lugar de referirnos explícitamente al tiempo de simulación, lo que se hace es indexar el tiempo, es decir, k se refiere al k–ésimo paso de integración.

Lo que se hizo fue convertir el sistema continuo anterior en un sistema de *tiempo discreto* asociado:

$$\mathbf{x_{k+1}} = \mathbf{F} \cdot \mathbf{x_k} \tag{2.12}$$

donde la matriz de evolución discreta $\mathbf{F}$ puede calcularse a partir de la matriz de evolución continua $\mathbf{A}$ y del paso de integración h, como:

$$\mathbf{F} = \mathbf{I}^{(\mathbf{n})} + \mathbf{A} \cdot h \tag{2.13}$$

El sistema discreto de la Ec.(2.12) es analíticamente estable si y sólo si todos sus autovalores se encuentran dentro de un círculo de radio 1 alrededor del origen, llamado *círculo unitario*. Para que esto ocurra, de la Ec.(2.13) se puede concluir que todos los autovalores de $\mathbf{A}$ multiplicados por el paso de integración h deben estar contenidos en un círculo de radio 1,0 alrededor del punto $-1{,}0$.

Se dice que un sistema lineal y estacionario de tiempo continuo integrado con un método dado de integración de paso fijo es *numéricamente estable* si y sólo si el sistema de tiempo discreto asociado es analíticamente estable.

Por lo tanto, al utilizar Forward Euler, un sistema analíticamente estable puede dar un resultado numéricamente inestable si el paso de integración es demasiado grande.

El algoritmo de BE tiene la ventaja de que si el sistema es analíticamente estable, se garantizará la estabilidad numérica para cualquier paso de integración h. Este método es entonces mucho más apropiado que el de FE para resolver problemas con autovalores alejados sobre el eje real negativo del plano complejo. Esto es de crucial importancia en los sistemas *stiff* (ver Sección 2.1.8), es decir, sistemas con autovalores cuyas partes reales están alejadas entre sí a lo largo del eje real negativo.

A diferencia de FE, en BE el paso de integración deberá elegirse exclusivamente en función de los *requisitos de precisión*, sin importar el *dominio de estabilidad numérica*.

2.1.4. Métodos Monopaso

Los métodos de Forward y Backward Euler realizan sólo aproximaciones de primer orden. Debido a esto, para obtener una buena precisión en la simulación, se debe reducir excesivamente el paso de integración lo que implica una cantidad de pasos y de cálculos en general inaceptable.

Para obtener aproximaciones de orden mayor, será necesario utilizar más de una evaluación de la función $\mathbf{f}(\mathbf{x}, t)$ en cada paso. Cuando dichas evaluaciones se realizan de manera tal que para calcular $\mathbf{x}_{k+1}$ sólo se utiliza el valor de $\mathbf{x}_k$, se dice que el algoritmo es *monopaso.* Por el contrario, cuando se utilizan además valores anteriores de la solución ($\mathbf{x}_{k-1}$, $\mathbf{x}_{k-2}$, etc.), se dice que el algoritmo es *multipaso.*

Los métodos monopaso se suelen denominar también *Métodos de Runge–Kutta*, ya que el primero de estos métodos de orden alto fue formulado por Runge y Kutta a finales del siglo XIX.

Métodos de Runge–Kutta

Los métodos explícitos de Runge–Kutta realizan varias evaluaciones de la función $\mathbf{f}(\mathbf{x}, t)$ en cercanías del punto $(\mathbf{x}_k, t_k)$ y luego calculan $\mathbf{x}_{k+1}$ realizando una suma pesada de dichas evaluaciones.

Un método de Runge–Kutta es un algoritmo que avanza la solución desde $x_k(t_k)$ hasta $x_{k+1}(t_k + h)$, usando una fórmula del tipo

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot (c_1 \cdot \mathbf{k_1} + \ldots + c_n \cdot \mathbf{k_n})$$

donde las llamadas etapas $k_1 \ldots k_n$ se calculan sucesivamente a partir de las ecuaciones:

$$\begin{array}{lrcl}
\text{etapa 0:} & \mathbf{k_1} & = & \mathbf{f}(\mathbf{x_k} + b_{1,1} \cdot h \cdot \mathbf{k_1} + \ldots + b_{1,n} \cdot h \cdot \mathbf{k_n}, t_k + a_1 h) \\
\vdots & \vdots & & \\
\text{etapa } n-1\text{:} & \mathbf{k_n} & = & \mathbf{f}(\mathbf{x_k} + b_{n,1} \cdot h \cdot \mathbf{k_1} + \ldots + b_{n,n} \cdot h \cdot \mathbf{k_n}, t_k + a_n h) \\
\text{etapa } n\text{:} & \mathbf{x_{k+1}} & = & \mathbf{x_k} + c_1 \cdot h \cdot \mathbf{k_1} + \ldots + c_n \cdot h \cdot \mathbf{k_n}
\end{array}$$

donde a_i y $b_{i,j}$ son constantes definidas por el método.

El número n de evaluaciones de función en el algoritmo se llama 'número de etapas' y frecuentemente es considerado como una medida del costo computacional de la fórmula considerada.

Métodos Monopaso Implícitos

Como vimos antes, el método de Backward Euler es un algoritmo implícito cuya principal ventaja es preservar la estabilidad de la solución numérica para cualquier paso de integración. Sin embargo, al igual que Forward Euler, realiza sólo una aproximación de primer orden.

Hay diversos métodos implícitos monopaso de orden mayor que, al igual que Backward Euler, preservan la estabilidad.

Uno de los más utilizados es la *Regla Trapezoidal.* Este método implícito realiza una aproximación de segundo orden y tiene la propiedad (al menos en sistemas

lineales y estacionarios) de que la solución numérica es estable si y sólo si la solución analítica es estable.

Si bien existen numerosos métodos implícitos monopaso, en la práctica no son tan utilizados ya que en general los métodos multipaso implícitos suelen ser más eficientes.

2.1.5. Algoritmos de Control de Pasos

Hasta aquí consideramos siempre el paso h como un parámetro fijo que debe elegirse previo a la simulación. Sin embargo, en muchos casos es posible implementar algoritmos que cambien el paso de integración de forma automática a medida que avanza la simulación.

Los algoritmos de *control de paso* tienen por propósito mantener el error de simulación acotado para lo cual ajustan automáticamente el paso en función del error estimado.

La idea es muy simple, en cada paso se hace lo siguiente:

1. Se da un paso con el método de integración elegido calculando $\mathbf{x}_{k+1}$ y cierto paso h.

2. Se estima el error cometido.

3. Si el error es mayor que la tolerancia, se disminuye el paso de integración h y se recalcula $\mathbf{x}_{k+1}$ volviendo al punto 1.

4. Si el error es menor que la tolerancia, se acepta el valor de $\mathbf{x}_{k+1}$ calculado, se incrementa el paso h y se vuelve al punto 1 para calcular $\mathbf{x}_{k+2}$.

La estimación del error se realiza generalmente con dos métodos de orden distintos y suponiendo que el de orden mayor da una aproximación con menos error se hace la diferencia entre los dos métodos.

2.1.6. Métodos Multipaso

Los métodos monopaso obtienen aproximaciones de orden alto utilizando para esto varias evaluaciones de la función $\mathbf{f}$ en cada paso. Para evitar este costo computacional adicional, se han formulado diversos algoritmos que, en lugar de evaluar repetidamente la función $\mathbf{f}$ en cada paso, utilizan los valores evaluados en pasos anteriores.

Los métodos implícitos multipaso más utilizados en la práctica son los denominados *Backward Differentiation Formula* (BDF). Por ejemplo, el siguiente es el método de BDF de orden 3:

$$\mathbf{x}_{k+1} = \frac{18}{11}\mathbf{x}_k - \frac{9}{11}\mathbf{x}_{k-1} + \frac{2}{11}\mathbf{x}_{k-2} + \frac{6}{11} \cdot h \cdot \mathbf{f}_{k+1} \tag{2.14}$$

Este método tiene prácticamente el mismo costo computacional que Backward Euler, ya que la ecuación a resolver es muy similar. Sin embargo, BDF3 es de tercer orden.

En los métodos multipaso se puede también controlar el paso de integración de manera similar a la de los métodos monopaso. Sin embargo, las fórmulas de los métodos multipaso son sólo válidas suponiendo paso constante (ya que usan

valores anteriores de la solución). Por lo tanto, para cambiar el paso en un método multipaso hay que interpolar los últimos valores de la solución a intervalos regulares correspondientes al nuevo paso de integración. En consecuencia, cambiar el paso tiene un costo adicional en este caso.

DASSL es un método de integración multi–paso muy popularizado basado en *Fórmulas de diferencia hacia atrás* o (Backward Differentiation Formula). Es el método utilizado por defecto en herramientas como OpenModelica y Dymola. Permite integrar también sistemas de ecuaciones diferenciales algebraicas (DAE).

2.1.7. Sistemas con Discontinuidades

Como se vio en las secciones anteriores de este capítulo, todos los métodos de integración de tiempo discreto se basan, explícita o implícitamente, en expansiones de Taylor. Las trayectorias siempre se aproximan mediante polinomios o mediante funciones racionales en el paso h en torno al tiempo actual t_k.

Esto trae problemas al tratar con modelos discontinuos, ya que los polinomios nunca exhiben discontinuidades, y las funciones racionales sólo tienen polos aislados, pero no discontinuidades finitas. Entonces, si un algoritmo de integración trata de integrar a través de una discontinuidad, sin dudas va a tener problemas.

Dado que el paso h es finito, el algoritmo de integración no reconoce una discontinuidad como tal. Lo único que nota es que la trayectoria de pronto cambia su comportamiento y actúa como si hubiera un gradiente muy grande.

La forma de evitar esto es en principio muy simple: lo que se necesita es un método de paso variable que dé un paso exactamente en el instante t^* en el que ocurre la discontinuidad. De esa forma, siempre se estará integrando una función continua antes de t^* y otra función continua después de t^*. Este es el principio básico de todos los métodos que realizan *manejo de discontinuidades*.

Las discontinuidades pueden clasificarse dentro de dos grandes categorías: Eventos temporales y Eventos de estado.

Eventos Temporales

Se denomina *Eventos temporales* a las discontinuidades de las cuales se sabe con cierta anticipación el tiempo de ocurrencia de las mismas. La forma de tratar eventos temporales es muy sencilla. Dado que se conoce cuándo ocurrirán, simplemente se le debe avisar al algoritmo de integración el tiempo de ocurrencia de los mismos. El algoritmo deberá entonces *agendar* dichos eventos y cada vez que dé un paso deberá tener cuidado de no saltearse ningún evento agendado. Cada vez que el paso de integración h a utilizar sea mayor que el tiempo que falta para el siguiente evento, deberá utilizar un paso de integración que sea exactamente igual al tiempo para dicho evento.

De esta manera, una simulación de un modelo discontinuo puede interpretarse como una secuencia de varias simulaciones continuas separadas mediante transiciones discretas.

Eventos de Estado

Muy frecuentemente, el tiempo de ocurrencia de una discontinuidad no se conoce de antemano. Lo que se sabe es la *condición del evento* en lugar del *tiempo del evento*.

Las condiciones de los eventos se suelen especificar como *funciones de cruce por cero*, que son funciones que dependen de las variables de estado del sistema y que se hacen cero cuando ocurre una discontinuidad.

Se dice que ocurre un *evento de estado* cada vez que una función de cruce por cero cruza efectivamente por cero. En muchos casos, puede haber varias funciones de cruce por cero.

Las funciones de cruce por cero deben evaluarse continuamente durante la simulación. Las variables que resultan de dichas funciones normalmente se colocan en un vector y deben ser monitoreadas. Si una de ellas pasa a través de cero, debe comenzarse una iteración para determinar el tiempo de ocurrencia del cruce por cero con una precisión predeterminada.

Así, cuando una condición de evento es detectada durante la ejecución de un paso de integración, debe actuarse sobre el mecanismo de control de paso del algoritmo para forzar una iteración hacia el primer instante en el que se produjo el cruce por cero durante el paso actual.

Una vez localizado este tiempo, la idea es muy similar a la del tratamiento de eventos temporales.

2.1.8. Sistemas Rígidos (Stiff)

Un sistema lineal y estacionario se dice que es stiff cuando es estable y hay estados que evolucionan muy rápidos y otros muy lentos. El problema con los sistemas stiff es que la presencia de los modos rápidos obliga a utilizar un paso de integración muy pequeño para que la simulación no se vuelva inestable a causa del método de integración.

Formalmente, un sistema de Ecuaciones Diferenciales Ordinarias se dice stiff si, al integrarlo con un método de orden n y tolerancia de error local de 10^{-n}, el paso de integración del algoritmo debe hacerse más pequeño que el valor indicado por la estimación del error local debido a las restricciones impuestas por la región de estabilidad numérica.

Para integrar entonces sistemas stiff sin tener que reducir el paso de integración a causa de la estabilidad, es necesario buscar métodos que incluyan en su región estable el semiplano izquierdo completo del plano $(\lambda \cdot h)$, o al menos una gran porción del mismo. Los métodos implícitos que hemos visto poseen esta propiedad por lo cual son aptos para simular sistemas rígidos.

2.2. Quantized State Systems (QSS)

La simulación del sistema (2.1) requiere usar métodos de integración numérica ya que no siempre hay una solución analítica. Como vimos previamente (Sección 2.1), mientras que los enfoques clásicos están basados en la discretización temporal una nueva familia de métodos de integración numérica fue desarrollada basada en la discretización de estado [21, 55].

Estos nuevos algoritmos, llamados Quantized State System methods (métodos de QSS), pueden aproximar Ecuaciones Diferenciales Ordinarias (ODE por sus siglas en inglés) como la de la Ec.(2.1) mediante modelos de eventos discretos.

Formalmente, el método de QSS de primer orden (llamado QSS1) aproxima la Ec. (2.1) por:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{v}(t)) \tag{2.15}$$

donde $\mathbf{q}$ es el *vector de estados cuantificados* y sus componentes están relacionadas una a una con las del vector de estados $\mathbf{x}$ siguiendo una función de cuantificación con histéresis:

$$q_j(t) = \begin{cases} x_j(t) \text{ si } |x_j(t) - q_j(t^-)| \geq \Delta Q_j \\ q_j(t^-) \text{ caso contrario} \end{cases} \tag{2.16}$$

. donde $q_j(t^-)$ es el límite por izquierda de q_j en t.

En la Figura 2.2 vemos la relación entrada-salida de una función de cuantificación de orden cero.

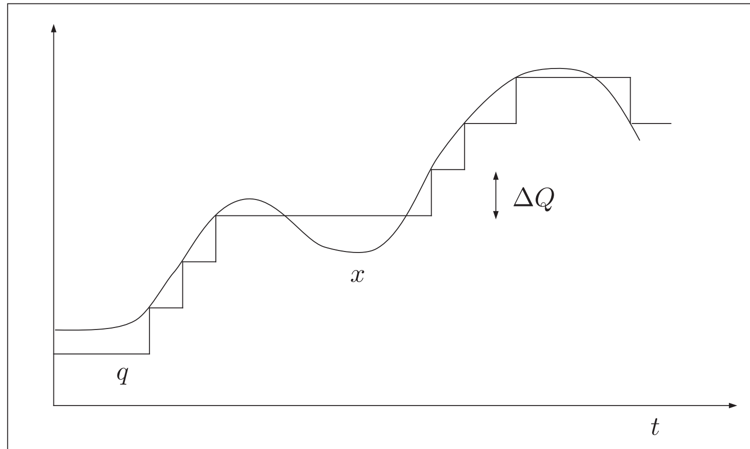

Figura 2.2: Función de Cuantificación con Histéresis de orden cero.

Notar que un cuantificador con histéresis tiene memoria. Es decir, calcula $q(t)$ no sólo en función del valor actual de $x(t)$ sino también de su valor pasado.

La presencia de esta función que relaciona $q_j(t)$ y $x_j(t)$ implica que $q_j(t)$ sigue una trayectoria constante a trozos que sólo cambia cuando difiere con $x_j(t)$ en un parámetro ΔQ_j llamado *quantum*.

Las variables q_j son llamadas variables cuantificadas y pueden ser vistas como una aproximación constante a trozos de la variable de estado correspondiente x_j.

De la misma forma las componentes de $\mathbf{v}(t)$ son aproximaciones constantes a trozos de las componentes correspondientes de $\mathbf{u}(t)$.

Los pasos de integración en los métodos de QSS sólo se producen cuando una variable cuantificada $q_j(t)$ cambia, esto es, cuando la variable de estado correspondiente $x_j(t)$ difiere de $q_j(t^-)$ en un quantum. Ese cambio implica también que algunas derivadas de estado (aquellas que dependen de x_j) también son modificadas. Luego, cada paso involucra un cambio en sólo una variable cuantificada y en algunas derivadas de estado.

Por lo tanto cuando un gran sistema ralo (o sparse) posee sólo actividad en unos pocos estados mientras que el resto del sistema se mantiene intacto, los métodos de QSS explotan intrínsecamente este hecho realizando cálculos sólo donde y cuando ocurren los cambios.

Otra ventaja importante de los métodos de QSS es que tratan las discontinuidades de una manera muy eficiente [53]. Dependiendo del orden del método, las variables de estado siguen trayectorias lineal a trozos, parabólica a trozos o constante a trozos. Debido a ello, detectar los cruces por cero es sencillo ya que se

debe resolver una ecuación cúbica en el peor caso. Una vez que la discontinuidad es detectada, el algoritmo la trata como un paso normal ya que *cada paso es de hecho una discontinuidad* en una variable cuantificada. Por lo tanto la ocurrencia de una discontinuidad implica sólo algunos cálculos locales para re-computar las derivadas de estados que están directamente afectadas por ese evento.

Estas ventajas resultan en una aceleración notable en el tiempo de simulación contra los algoritmos de integración numérica clásicos. En modelos con discontinuidades frecuentes como sistemas de electrónica de potencia, los métodos de alto orden que veremos a continuación, pueden simular hasta 20 veces más rápido que los métodos convencionales [53].

Métodos QSS de Alto Orden

Basados en QSS1, se desarrollaron métodos de alto orden QSS2 [52] y QSS3 [54], que realizan aproximaciones de segundo y tercer orden respectivamente.

El método QSS2 esta basado en el mismo principio que QSS1 pero reemplaza la función de cuantificación de cero–orden de la Ec. (2.16) por una función de cuantificación de primer orden

El comportamiento de entrada-salida de una función de cuantificación de primer orden se muestra en la Figura 2.3.

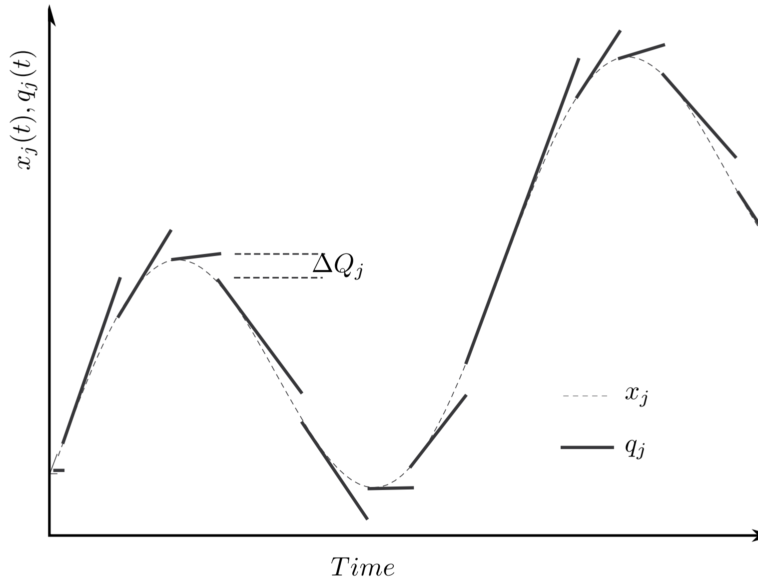

Figura 2.3: Función de cuantificación de primer orden.

La definición del método de QSS2 es idéntico a la de QSS1. Esto es, QSS2 aproxima la Ec. (2.1) por la Ec.(2.15), excepto que utiliza una función de primer orden para relacionar x_j y q_j.

Para obtener un método de tercer orden no sólo la primera derivada sino también la segunda derivada de las trayectorias del sistema debe ser tomadas en cuenta. Para ello la función de cuantificación de primer orden de la Figura 2.3 es reemplazada por una función de cuantificación de *segundo orden* como muestra la Figura

2.4.

Una función de cuantificación de segundo orden genera en su salida una trayectoria parabólica a trozos la cual posee valor, pendiente y segunda derivada. Cuando la salida difiere de la entrada en más de un quantum la función de cuantificación de segundo orden emite como salida un nuevo tramo de parábola con los valores actualizados.

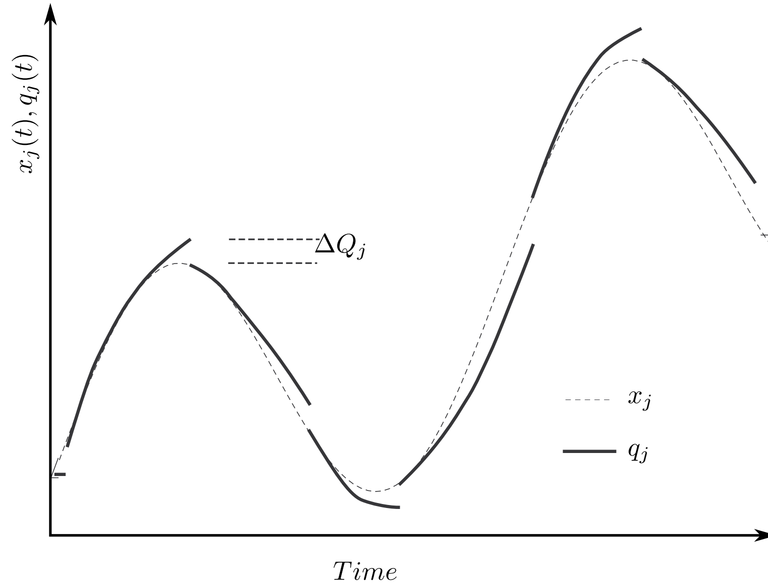

Figura 2.4: Función de cuantificación de segundo orden.

La definición del método de QSS3 es idéntica a las de QSS1 y QSS2. QSS3 aproxima Ec. (2.1) por Ec. (2.15), excepto que utiliza una función de cuantificación de segundo orden para relacionar x_j y q_j.

2.2.1. Propiedades Teóricas de los métodos de QSS

Definiendo $\mathbf{\Delta x}(t) \triangleq \mathbf{x}(t) - \mathbf{q}(t)$, la aproximación de QSS (Ec. (2.15)) puede ser re–escrita como:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t) + \mathbf{\Delta x}(t), \mathbf{u}(t)) \tag{2.17}$$

Esta ecuación es idéntica a la ODE original de la Ec. (2.1) excepto por el término de perturbación $\mathbf{\Delta x}(t)$.

Una propiedad fundamental de la función de cuantificación es:

$$|x_j(t) - q_j(t)| \leq \Delta Q_j \quad \forall t \geq 0 \tag{2.18}$$

Tomando en cuenta la Ec. (2.18), resulta que cada componente de esta perturbación esta acotada por el quantum correspondiente ΔQ_j.

Basado en esta observación se ha probado que:

- La solución numérica obtenida por el método de QSS1 converge a la solución analítica [55].

- Si la ODE original tiene un punto de equilibrio estable asintótico, la solución de QSS1 estará acotada dentro de ese punto de equilibrio [55].

- En un sistema Lineal Invariante al Tiempo (o LTI), el error global cometido por QSS1 está acotado. Una cota superior para el error puede ser calculada, la cual depende linealmente en la cuantificación (i.e. el quantum) utilizada [52].

Como QSS2 y QSS3 comparten con QSS1 la propiedad de la Ec. (2.18), ellos también comparten las propiedades antes descriptas.

La familia de métodos de QSS se completa con cuatro métodos más para sistemas stiff llamados Backward QSS y Linearly Implicit QSS de orden 1, 2 y 3 (BQSS, LIQSS, LIQSS2, LIQSS3, respectivamente) [63, 64] y un método para sistemas marginalmente estables (Centered QSS [22] o CQSS).

Todos estos métodos tienen la característica de ser explícitos, lo que constituye una ventaja ante los métodos tradicionales sobre todo pensando en aplicaciones de tiempo real ya que no involucran un proceso iterativo.

2.2.2. Cuantificación Logarítmica

El uso de un quantum uniforme conlleva a controlar el *error absoluto*. En muchos casos es preferible controlar el *error relativo* de la simulación, ya que implica ajustar la precisión con la magnitud (absoluta) de la variable en cuestión.

En los métodos de QSS, esto puede ser realizado utilizando un quantum de tamaño proporcional a la magnitud de la variable de estado correspondiente. Esta idea lleva a una *cuantificación logarítmica* [51], y se ha mostrado que su uso implica intrínsecamente controlar el error relativo.

Una función de cuantificación logarítmica se especifica definiendo un *quantum relativo*, ΔQ_{rel}, y un *quantum mínimo*, ΔQ_{min}. Luego, el quantum se modifica con la variable cuantificada correspondiente q_j de acuerdo a la ecuación:

$$\Delta Q_j = \text{máx}(\Delta Q_{rel} \cdot |q_j|, \Delta Q_{min})$$

2.3. Discrete EVent Systems (DEVS)

DEVS [94] es un formalismo para modelar y analizar sistemas de eventos discretos (es decir, sistemas en los cuales en un lapso finito de tiempo, ocurren una cantidad finita de eventos).

Un modelo DEVS puede ser visto como un autómata que procesa una serie de eventos de entrada y genera una serie de eventos de salida. Este procesamiento está regido por la estructura interna de cada una de las partes que componen el modelo general.

Un modelo DEVS está descripto por dos clases de componentes, modelos atómicos y modelos acoplados.

2.3.1. Modelos DEVS Atómicos

Un modelo atómico representa la unidad "indivisible" de especificación, en el sentido que es la pieza fundamental y más básica de un modelo DEVS. Formalmente un modelo atómico está conformado por la 7-upla:

$$(X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta) \text{ donde:}$$

- X es el conjunto de valores de entrada que acepta el modelo atómico, es decir un evento de entrada tiene como valor un elemento del conjunto X.
- Y es el conjunto de valores de los eventos de salida que puede emitir el modelo atómico.
- S es el conjunto de estados internos del modelo, en todo momento el atómico está en un estado dado, que es un elemento del conjunto S.
- ta es una función $S \to R^+$, que indica cuánto tiempo el modelo atómico permanecerá en un estado dado, si es que no se recibe ningún evento de entrada. Esta función puede asociarse también al tiempo de vida de un estado.
- δ_{int} es una función $S \to S$, que indica la dinámica del sistema en el momento que el modelo atómico realiza una transición interna. Sería el análogo a una tabla de transición en otros autómatas.
- δ_{ext} es una función $(S \times \mathbb{R}^+ \times X) \to S$, que indica el cambio de estado ante la presencia de un evento externo.
- λ es una función $S \to Y$ que indica qué evento se debe emitir al salir de un estado dado.

Los conjuntos S, X e Y son arbitrarios, y en general infinitos, a diferencia de lo que ocurre con los autómatas de estados finitos y otros formalismos similares.

Cada posible estado s ($s \in S$) tiene asociado un *Avance de Tiempo* calculado por la *Función de Avance de Tiempo* $ta(s)$.

En la Figura 2.5 vemos la evolución de un modelo atómico. Si el estado toma el valor s_1 en el tiempo t_1, tras $ta(s_1)$ unidades de tiempo (o sea, en tiempo $ta(s_1)+t_1$) el sistema realizará una *transición interna* yendo a un nuevo estado s_2 dado por $s_2 = \delta_{int}(s_1)$. La función δ_{int} se llama *Función de Transición Interna*.

Cuando el estado va de s_1 a s_2 se produce también un evento de salida con valor $y_1 = \lambda(s_1)$. La función λ ($\lambda : S \to Y$) se llama *Función de Salida*. Así, las funciones ta, δ_{int} y λ definen el comportamiento autónomo de un modelo DEVS.

Cuando llega un evento de entrada, el estado cambia instantáneamente. El nuevo valor del estado no sólo depende del valor del evento de entrada sino también del valor anterior del estado y del tiempo transcurrido desde la última transición.

Si el sistema llega al estado s_3 en el instante t_3 y luego llega un evento de entrada en el instante $t_3 + e$ con un valor x_1, el nuevo estado se calcula como $s_4 = \delta_{ext}(s_3, e, x_1)$ (notar que $ta(s_3) > e$). En este caso se dice que el sistema realiza una *transición externa*. La función δ_{ext} se llama *Función de Transición Externa*. Durante una transición externa no se produce ningún evento de salida.

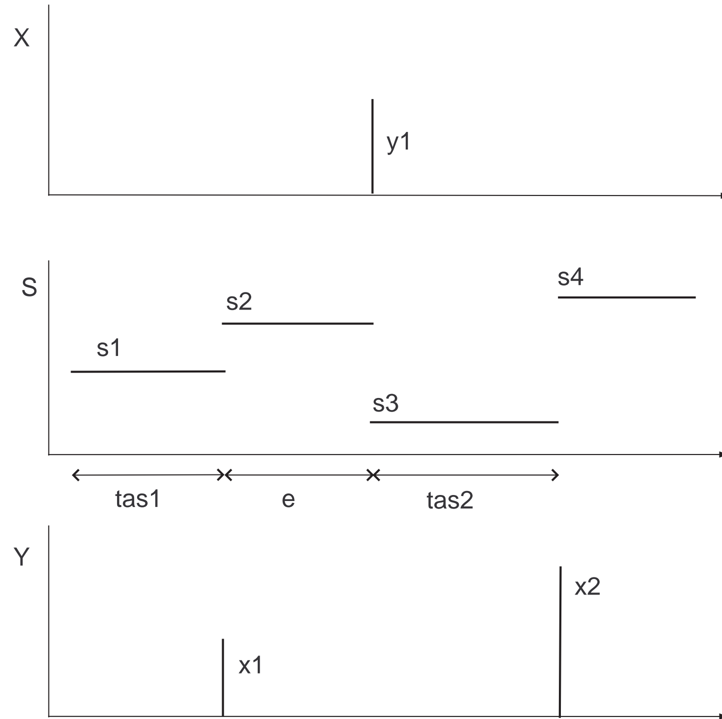

Figura 2.5: Comportamiento de un modelo DEVS atómico

2.3.2. Modelos DEVS Acoplados

La descripción de un sistema puede ser completamente realizada utilizando modelos atómicos, aunque esto resulta un poco incómodo y confuso. Los conjuntos de estados y las funciones de transición se vuelven inmanejables en sistemas complejos, y nunca podemos asegurar haber cubierto todos los posibles estados. Para abordar este problema, el formalismo DEVS introduce lo que se llaman *modelos acoplados* (ver Figura 2.6) que es una forma de agrupar modelos DEVS y generar nuevos modelos a partir de este agrupamiento.

Hay dos formas de acoplamiento, la más general, en la cual se utilizan funciones de traducción entre los sub–sistemas y otra clase que adopta el uso de puertos para la comunicación entre sub–sistemas. Aunque estas dos formas son equivalentes entre sí, describiremos la segunda clase, ya que es la más simple y es la utilizada en el presente trabajo.

Formalmente un modelo acoplado está representado por la octo-upla:

$$N = (X_N, Y_N, D, \{M_d\}, EIC, EOC, IC, Select)$$

donde cada componente es:

- X_N es el conjunto de eventos de entrada al modelo acoplado, representado por el producto cartesiano del conjunto de puertos de entrada $InPorts$ y el conjunto de posibles valores para cada puerto. O sea un evento de entrada

al modelo acoplado está representado por un par (p, v) donde $p \in InPorts$ y $v \in X_p$.

- Y_N es el conjunto de eventos que el modelo puede emitir. Es un elemento del producto cartesiano entre el conjunto de puertos de salida $OutPorts$ y el conjunto de posibles valores para este puerto, o sea un evento de salida del modelo acoplado está representado por un par (p, v) donde $p \in OutPorts$ y $v \in Y_p$.

- D es el conjunto de los índices a los modelos DEVS (atómicos y acoplados) que conforman este modelo.

- $\{M_d\}$ es el conjunto de los modelos atómicos y/o acoplados (son justamente los modelos que "acopla" o "agrupa" este modelo acoplado).

- EIC y EOC estos conjuntos indican la forma que los modelos internos al acoplado se relacionan con el exterior:

  - EIC (o External Input Coupling) son las conexiones de entrada al acoplado, es decir, conecta un puerto de entrada del acoplado con un puerto de entrada de un modelo perteneciente al acoplado.
  - EOC (o External Output Coupling) son las conexiones de salida del acoplado. Conecta un puerto de salida de un modelo interno del acoplado con un puerto de salida del acoplado. Formalmente:

    $$\begin{aligned} EIC &\in \{((N, ip_N), (d, ip_d))|ip_N \in InPorts, d \in D, ip_d \in InPorts_d\} \\ EOC &\in \{((d, op_d), (N, op_N))|op_N \in OutPorts, d \in D, op_d \in OutPorts_d\} \end{aligned}$$

    donde N es el modelo acoplado.

- IC representa las conexiones internas del acoplado donde:

  $$IC \in \{((a, ip_a), (b, ip_b))|a, b \in D, ip_a \in OutPorts_a, ip_b \in InPorts_b\}$$

  donde no se permite que $a = b$.

- $Select$ es una función $(D \times D \to D)$ que decide qué modelo realizará primero su transición interna, si se da el caso de eventos simultáneos. Es una función de "desempate" que en ciertos modelos es necesaria.

$InPorts$ y $Outports$ son conjuntos que describen los posibles puertos de entrada y salida respectivamente. En general se utilizan números enteros para representar los puertos posibles por lo cual $InPorts = \mathbb{N}$ y $Outports = \mathbb{N}$ Los modelos acoplados son en sí mismos modelos DEVS válidos; formalmente el acoplamiento (como lo definimos antes) es una operación cerrada sobre el conjunto de modelos DEVS. Acoplar modelos DEVS forma nuevos modelos DEVS.

Sin esta cualidad el acoplamiento resultaría inútil desde del punto de vista del formalismo. También trae muchas ventajas a la hora de describir modelos DEVS y a la hora de simularlos. El acoplamiento da lugar a una estructura jerárquica de desarrollo.

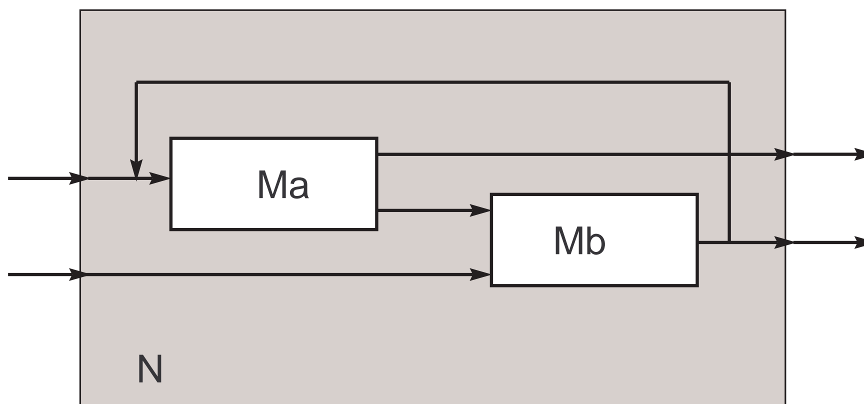

Figura 2.6: Modelo acoplado

2.3.3. Simulación de Modelos DEVS

La simulación de modelos DEVS puede ser llevada a cabo fácilmente en cualquier computadora. Describiremos aquí, un algoritmo para realizar tal que puede ser implementado en cualquier lenguaje de programación actual.

Para simular el funcionamiento de un sistema DEVS sólo debemos realizar los siguientes pasos:

1. Buscar el modelo (perteneciente al modelo acoplado raíz[1], o sea buscar un $d \in D$), que de acuerdo a su función de avance ta y el tiempo transcurrido desde su última transición, es el próximo a realizar una transición interna. A este modelo, lo llamaremos d^* y t_n al tiempo de su próxima transición interna (la indicada por ta).

2. Avanzamos el tiempo de simulación t a t_n y ejecutamos la transición interna del modelo d^*.

3. Luego propagamos el evento de salida producido por la transición interna realizada por d^*, a los modelos conectados a él, ejecutando a su vez la transición externa de cada uno de éstos.

4. Finalmente volvemos a realizar el ciclo desde el primer paso.

Una manera sencilla de implementar estos pasos es escribir un programa que reproduzca la estructura jerárquica del modelo a simular. Este método es el desarrollado en [94] donde una rutina llamada *DEVS-simulator* es asociada a cada *Modelo Atómico* y una rutina diferente llamada *DEVS-coordinator* es asociada a cada *Modelo Acoplado*. En la cima de esta jerarquía hay una rutina llamada *DEVS-root-coordinator* la cual trata con el tiempo global de simulación.

La Figura 2.7 ilustra esta idea sobre un modelo DEVS Acoplado.

[1]Aquí suponemos que siempre hay un modelo acoplado en el nivel superior del modelo. De no ser así todos los modelos de nivel superior pueden ser embebidos en un modelo acoplado (sin puertos de entrada, ni de salida).

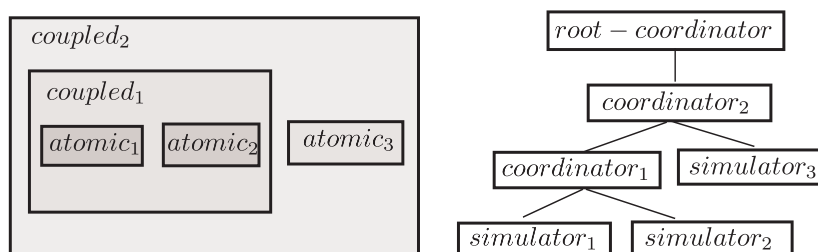

Figura 2.7: Estructura jerárquica de un modelo y de su simulación

Los simuladores y coordinadores de capas consecutivas se comunican a través de mensajes. Los coordinadores envían mensajes a sus hijos para que ellos ejecuten las funciones de transición. Cuando un simulador ejecuta una transición, calcula su próximo estado y emite un evento de salida (si fuera una transición interna) En todos los casos el estado del simulador coincidirá con el estado del modelo DEVS atómico asociado.

Cuando un coordinador ejecuta una transición, envía mensajes a alguno de sus hijos para que ellos ejecuten la transición correspondiente. Cuando un evento de salida es producido por alguno de sus hijos, este evento debe ser propagado a los hijos y al exterior del acoplado si fuera necesario.

Cada simulador o coordinador tiene una variable local tn la cual indica el tiempo de su próxima transición interna. En los simuladores esa variable es calculada utilizando la función de avance $ta(s)$ del modelo atómico correspondiente. En los coordinadores es calculada como el mínimo tn de sus hijos. Por lo tanto, el tn del coordinador en la cima es el tiempo en el cual ocurrirá el próximo evento del sistema completo. El *root-coordinator* sólo mira a este tiempo tn, avanza el tiempo de simulación t a este valor y envía un mensaje para que sus hijos realicen la transición. Luego repite este ciclo hasta finalizar la simulación.

2.3.4. Herramientas de Simulación DEVS

Existen diversas herramientas de modelado y simulación DEVS. Algunas de ellas son sólo librerías de simulación donde el usuario debe encargarse de programar el modelo mientras que otras poseen un editor gráfico.

La primera categoría incluye adevs [73], una librería en C++ que permite utilizar dos extensiones del formalismo DEVS, Parallel DEVS [24] y Dynamic DEVS [48]. DEVSJAVA y DEVS/HLA [94] se encuentran también en esta categoría. Ambas herramientas están escritas en Java y soportan la ejecución en multiprocesadores y en tiempo real.

CD++ [89] es una herramienta para modelado y simulación DEVS que implementa también una extensión llamada Cell–DEVS la cual se basan en la unión de DEVS y autómatas celulares. Una característica de CD++ es que su ejecución puede sera realizada en tiempo real.

Por último PowerDEVS es un entorno de modelado y simulación DEVS orientado a sistemas continuos e híbridos que veremos en detalle en la Sección 2.5.

2.4. Formalismo DEVS y QSS

Aunque los métodos de QSS pueden ser escritos en cualquier lenguaje de programación, su implementación se simplifica mucho si uno los describe en el formalismo DEVS modelando cada componente como un modelo DEVS.

Como veremos a continuación, DEVS provee un entorno unificado para representar las dinámicas continuas y discretas y acoplarlas en un solo modelo.

Como vimos en la Sección 2.2, un modelo continuo descripto por una ODE puede ser representado mediante QSS por la ecuación (2.15). Esta representación puede ser exactamente simulada por un modelo DEVS.

Debido a que las componentes de $q_j(t)$ y de $v_j(t)$ siguen una trayectoria constante a trozos, las derivadas de los estados $\dot{x}_j(t)$ también seguirán trayectorias constante a trozos. Luego, las variables de estado $x_j(t)$ tienen una evolución lineal a trozos.

Cada componente de la Ec. (2.15) puede ser vista como el acoplamiento de dos subsistemas elementales, uno estático,

$$\dot{x}_j(t) = f_j(q_1, \cdots, q_n, v_1, \cdots, v_m) \tag{2.19}$$

y uno dinámico

$$q_j(t) = Q_j(x_j(\cdot)) = Q_j(\int \dot{x}_j(\tau)d\tau) \tag{2.20}$$

donde Q_j es la función de cuantificación (notar que no es una función instantánea del valor de $x_j(t)$ sino una función de la trayectoria $x_j(\cdot)$).

Como las componentes de $v_j(t)$, $q_j(t)$ y $\dot{x}_j(t)$ son constantes a trozos, ambos sub–sistemas pueden ser representados por una secuencia de eventos.

El modelo DEVS será el acoplamiento de n integradores cuantificados HQI (utilizando la cuantificación de QSS), n funciones estáticasF_i y m fuentes de señales $v(t)$. De hecho el modelo DEVS resultante luce idéntico al diagrama en bloques del sistema original de la Ec. (2.1) como lo muestra la Figura 2.8.

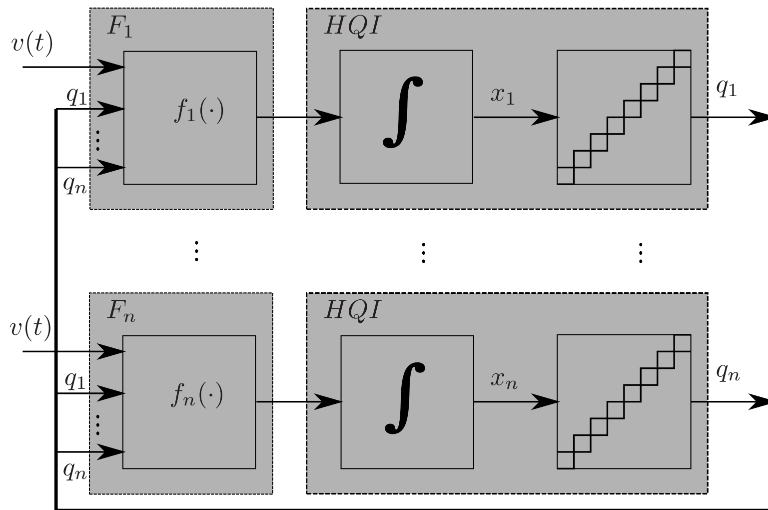

Figura 2.8: Modelo DEVS acoplado de una aproximación QSS

Si utilizamos una función de cuantificación de orden cero, la función estática j–ésima calcula una trayectoria constante a trozos $\dot{x}_j(t)$ a partir de las trayectorias constante a trozos de los estados cuantificados q_i de acuerdo a $\dot{x}_j = f_j(\mathbf{q}, t)$. Como dijimos, estas trayectorias son representables mediante una secuencia de eventos dentro del formalismo DEVS.

Similarmente, el integrador j–ésimo integra la trayectoria constante a trozos $\dot{x}_j$ calculada por la función estática correspondiente y computa la trayectoria q_j constante a trozos.

Los modelos atómicos DEVS para las funciones estáticas y para los integradores cuantificados son bastante simples y han sido descriptos en [21, 54]. Acoplando estos modelos DEVS obtenemos un nuevo modelo que puede ser utilizado para simular el comportamiento de la aproximación de QSS1 de la ODE original.

Los métodos de alto orden de QSS pueden ser implementados de la misma forma. Para este fin, los eventos de los integradores cuantificados y de las funciones estáticas representarán los coeficientes de las trayectorias polinomiales correspondientes.

Toda la familia de métodos QSS ha sido implementada e incluida en la librería de PowerDEVS [11] que veremos a continuación.

2.5. Herramienta de Simulación PowerDEVS

PowerDEVS [11] es un entorno de simulación de modelos DEVS de propósito general. Posee varias características que lo hacen apto para la simulación de sistemas híbridos y para la implementación en tiempo real. Los modelos atómicos se programan en C++, aunque en principio, el entorno gráfico podría ser utilizado para cualquier lenguaje de desarrollo.

La implementación de esta herramienta está dividida en dos módulos:

Interfaz Gráfica (GUI): Es la interfaz que ve el usuario regular de PowerDEVS (ver Figura 3.5). Permite describir modelos DEVS de una forma gráfica y sencilla. Los modelos acoplados son descriptos arrastrando y conectando sub–modelos mientras que los modelos atómicos son descriptos en C++. La interfaz está implementado en Qt/C++ y corre bajo diversos sistemas operativos (Unix/Windows). La GUI es la encargada de generar el código C++ que luego será compilado y ejecutado para simular el modelo DEVS correspondiente.

Motor de simulación: Este módulo es el núcleo de toda la herramienta. El código que es generado por la GUI luego es compilado junto con el motor de simulación y se obtiene un programa ejecutable que simula el modelo DEVS. Su implementación está hecha en C++ por lo cual puede ser portado fácilmente a cualquier arquitectura y sistema operativo. Básicamente es una implementación del método de simulación de modelos DEVS descripto en [94].

Una de las principales características de PowerDEVS es que posee una completa implementación de los métodos de integración de QSS para la simulación de sistemas híbridos.

Aunque la distribución estándar de PowerDEVS provee varios modelos para la simulación de sistemas híbridos, el usuario puede agregar nuevos bloques. Para ello

el usuario debe proveer una descripción en C++ del comportamiento del nuevo bloque. Durante la simulación del modelo el motor de simulación de PowerDEVS creará instancias de estas clases y las conectará de la forma que el usuario ha descripto en la GUI.

2.6. Sistemas de Tiempo Real

Un sistema de tiempo real, en el sentido más general, es aquel en el cual el resultado obtenido no sólo debe ser correcto sino que debe ser también "entregado" en el momento correcto [85] y que responde a estímulos cumpliendo también restricciones temporales. Los eventos y cálculos internos tienen un *deadline*[1] y el sistema es el encargado de realizar estos cálculos de forma que se cumplan estas restricciones.
Dentro de los sistemas de tiempo real tenemos dos grandes clases:

Sistemas de tiempo real *hard*: son aquellos en los cuales la "pérdida" de un *deadline* implica un error irrecuperable en el sistema o vuelve el resultado obtenido inútil. Ejemplos de estos sistemas son el sistema de frenado ABS (Anti-Block-System) de un auto (una señal enviada tardíamente puede resultar en graves daños), un marcapasos, o un sistemas de control industrial.

Sistemas de tiempo real *soft*: son aquellos en los cuales la pérdida de un *deadline* puede ser superada o ignorada causando quizás un resultado de "menor calidad". Entre estos sistemas se encuentran, transferencia de audio o video (la falta de un fragmento de audio o video puede ser subsanada o en el peor caso ignorada), sistemas que realizan tareas de mantenimiento (por ejemplo los servicios o demonios en Linux).

No debe confundirse tiempo real con alta performance. Un sistema puede tener un alto rendimiento y no ser capaz de cumplir restricciones temporales y por el contrario otro sistema puede tener un rendimiento menor y sí cumplir restricciones temporales. En general los dispositivos electrónicos que se utilizan para implementar sistemas de tiempo real, son dispositivos dedicados o embebidos en los cuales se puede "predecir" su tiempo de respuesta ante los estímulos y puede diagramarse el sistema de una forma simple.

2.6.1. Sistemas Operativos de Tiempo Real (RTOS)

Hoy en día, muchos sistemas de tiempo real son implementados en computadoras de uso general. Para ello, se han desarrollado sistemas operativos de tiempo real (RTOS) que abstraen y manejan el hardware subyacente brindando al usuario las herramientas para desplegar el sistema con restricciones temporales. Estos RTOS permiten describir las restricciones temporales que posee cada tarea y el sistema se encargará de planificarlas de modo que estas se cumplan. Existen diversas políticas de planificación de procesos, entre ellas *Round-Robin*, *Earliest Deadline First*, *Rate Monotonic*. Más información sobre sistemas operativos de tiempo real puede ser encontrada en [85].

[1] El deadline es el momento en el cual debe ser entregado el resultado.

Dentro de los RTOS más popularizados encontramos a QNX [44], FreeRTOS [59], RTAI [26], Xenomai [38], VxWorks[92], RTLinux [4]. Cabe mencionar que ni Windows ni Linux (en sus versiones estándar) son RTOS.

En el marco de este trabajo decidimos usar RTAI para nuestra implementación por las siguientes razones:

- Posee licencia GPL lo cual nos permite utilizarlo sin costo alguno (y si fuese necesario modificarlo).

- Es una extensión de GNU/Linux, por lo tanto todo el bagaje de herramientas pre–existentes (PowerDEVS, compilador C++ (g++), Scilab, GNUPlot, etc) ejecutará sin modificación en RTAI (aunque *no en tiempo real*).

- Permite al usuario definir las tareas de tiempo real como procesos de usuario. Esta característica es muy útil ya que otros RTOS obligan al usuario a escribir un módulo de núcleo lo cual se torna dificultoso de implementar y depurar.

2.6.2. RealTime Application Interface – RTAI

La RealTime Application Interface (RTAI) es un RTOS portado a diversas arquitecturas (i386, PowerPC, ARM). Extiende al núcleo de Linux para dar soporte a tareas de tiempo real que se ejecutarán concurrentemente con los procesos de Linux normales.

Como dijimos antes Linux no es de por sí un RTOS. No provee servicios de tiempo real y deshabilita las interrupciones de hardware en ciertas partes del núcleo para asegurar atomicidad. Estos períodos en los cuales no pueden ocurrir interrupciones llevan a un escenario donde el tiempo de respuesta del sistema es desconocido y los deadlines pueden ser violados.

Para evitar esto, RTAI inserta un micro–núcleo por debajo del núcleo de Linux. De esta forma Linux ejecuta como si RTAI no existiera y este micro–núcleo es el encargado de capturar y manejar las interrupciones.

Cuando se ejecuta un programa en Linux el scheduler puede quitarle el procesador (preempting) en cualquier momento. Esto es inaceptable en un sistema de tiempo real. Por lo tanto, en RTAI todas las tareas de tiempo real son ejecutadas sobre el micro–núcleo sin la supervisión de Linux. De hecho las tareas de tiempo real no son vistas por el scheduler de Linux. Éstas son manejadas por el scheduler de RTAI. Luego, en un sistema RTAI hay dos clases de procesos, los procesos normales de Linux y los de tiempo real de RTAI. Los segundos no puede hacer uso de los servicios provistos por Linux (como el sistema de archivos) ya que esto podría violar los deadlines. Para solucionar esto, RTAI ofrece varios mecanismos de comunicación entre procesos (o IPC).

RTAI ofrece los siguientes servicios para implementar sistemas en tiempo real.

Tiempo de respuesta de interrupción acotado: Una interrupción es un evento externo al sistema. El tiempo de respuesta a este evento variará de computadora a computadora pero RTAI garantiza un límite superior (en cada computadora) el cual es necesario para comunicarse con hardware externo, como por ejemplo una placa adquisidora de datos.

Inter Process Communication (IPC): RTAI ofrece diferentes métodos de comunicación entre procesos como semáforos, pipes, mailboxes, memoria compartida, entre otros. Estos mecanismos de IPC pueden ser utilizados para comunicar tareas de tiempo real de RTAI con procesos de Linux.

Relojes de alta precisión: Cuando se desarrolla un sistema de tiempo real la precisión con la cual se maneja el tiempo es muy importante. RTAI posee relojes y funciones de espera con granularidad de nanosegundos (1.0e-9 seg.).

Manejo de interrupciones y acceso al hardware: RTAI permite capturar las interrupciones de hardware y tratarlas con un manejador definido por el usuario. Permite también el acceso a puertos y dispositivos de bajo nivel.

Soporte para multiprocesadores RTAI tiene soporte para arquitecturas multi–core permitiendo al usuario correr simultáneamente distintas tareas de tiempo real en procesadores separados.

2.7. Simulación en Paralelo de Eventos Discretos

La computación en paralelo es una clase de cómputo donde los cálculos son realizados simultáneamente en diferentes procesadores. En ciertas ocasiones, problemas grandes pueden ser divididos en sub–problemas más pequeños que pueden ser resueltos concurrentemente.

En particular, la simulación en paralelo de eventos discretos o *Parallel Discrete–Event Simulation* (PDES) es el área de investigación que estudia la ejecución de sistemas de eventos discretos en computadoras paralelas. Explotando el paralelismo potencial en el modelo, PDES puede superar las limitaciones de una simulación secuencial tanto en tiempo de simulación como en consumo de recursos. Por lo tanto, PDES ofrece grandes beneficios al simular sistemas de gran escala que demandan una inmensa cantidad de recursos computacionales.

Hay diversas formas de particionar la simulación en varias sub–simulaciones. La más simple es la denominada "repetición de ensayos" [43] (o *parameter sweeping*), la cual ejecuta múltiples ejecuciones de una simulación secuencial concurrentemente. La desventaja de esta forma es que no ofrece ninguna aceleración en cada simulación secuencial. Otra forma es la llamada "paralelización temporal" [42] donde la simulación se divide en tajadas de tiempo y cada simulación computa un período de tiempo distinto. Debido a la dependencia entre estado inicial de una simulación y estado final de otra, la ganancia de este enfoque depende directamente de si el modelo relajado (es decir divido en distintas simulaciones) obtiene un resultado correcto.

Finalmente, la técnica llamada "espacio paralelo" [57] (basada en descomposición de funcional) divide al sistema en una colección de sub–sistemas, donde cada uno es simulado en un Procesador Lógico. Esta división puede ser realizada por el modelador de forma manual o en algunos casos de forma automática mediante un algoritmo. Cada Procesador Lógico mantiene su tiempo de simulación propio y sólo es capaz de procesar eventos del sub–sistema que tienen designado. La comunicación entre Procesadores Lógicos se realiza exclusivamente intercambiando mensajes que contienen eventos junto con un tiempo de ocurrencia, también llamado eventos con *time–stamp* . Esta técnica es en general más robusta que otras técnicas de paralelización, debido a que la descomposición de datos es aplicable

naturalmente a la mayor parte de los models. En este trabajo trataremos con esta clase de paralelización.

2.7.1. Plataformas de Paralelización

A la hora de implementar PDES en un sistema real debemos ejecutar cada Procesador Lógico en un procesador físico. Cada uno de estos procesadores físicos ejecutará un código distinto con entrada distinta, esto es, de acuerdo a la taxonomía de Flynn [31], operaremos con arquitecturas de la clase Multiple Instructions and Multiple Data (MIMD).

En particular, en la presente Tesis trabajaremos con arquitecturas de procesadores multi-core [15] ya que nos ofrecen muchas ventajas para implementar MIMD, como memoria compartida, comunicación local, y un juego de instrucciones genérico.

Actualmente se está realizando mucho trabajo con procesadores gráficos (GPGPU) para cómputos científicos [17, 56, 67, 68]. Aunque esta plataforma ofrece muchos procesadores físicos a bajo costo, tiene sus desventajas. El uso de memoria compartida entre los procesadores conlleva una gran latencia, los caminos divergentes en el código no son eficientes, la sincronización entre hilos es costosa y el juego de instrucciones no es tan general como el de una arquitectura de propósito general. Esto no quiere decir que GPGPU sea una mala idea sino sólo que no son convenientes para esta clase de aplicaciones.

Por otro lado, arquitecturas basadas en multi-computadoras (o *clusters*) con paso de mensajes, como por ejemplo MPI [82], carecen de memoria compartida lo cual hace más costosa la transmisión de eventos entre simulaciones. La sincronización también conlleva un costo.

2.7.2. Algoritmos de Sincronización

Al simular sistemas de eventos discretos, los eventos deben ser procesados en un orden no decreciente de time–stamp ya que un evento con menor time–stamp podría cambiar el estado del sistema, afectando los eventos subsiguientes. Esto se denomina restricción causal. Si los eventos simultáneos son ordenados (eventos con el mismo time–stamp) determinísticamente usando algún método de desempate, la restricción causal impone un orden total sobre los eventos.

Cómo vimos previamente, en PDES, cada Procesador Lógico tendrá su lista de eventos propia y mantendrá su propio tiempo de simulación. Cuando un evento debe ser enviado a otro Procesador Lógico se le enviará con el time–stamp propio del Procesador Lógico. Este time–stamp podría ser menor que el tiempo de simulación del receptor lo cual violaría la restricción causal. Por lo tanto en PDES, el problema fundamental es preservar la causalidad en cada Procesador Lógico siendo que cada uno tiene su propio tiempo de simulación.

Presentamos aquí un resumen (basado en [57]) de los algoritmos de sincronización en la literatura previos a esta Tesis.

Sincronización CMB

Chandy, Misra [23] y Bryant [19] (CMB) presentaron a finales de los 70's el primer algoritmo de sincronización para la simulación en paralelo de sistemas de

eventos discretos. Este algoritmo supone que los Procesadores Lógicos intercambian eventos (con time–stamp) que serán transmitidos en orden cronológico (es decir con time–stamp crecientes). Para este fin, cada Procesador Lógico mantiene una cola por cada conexión con otro Procesador Lógico donde almacena los eventos que recibe desde ese Procesador Lógico. También se le asocia a cada cola un valor de reloj igual al mínimo time–stamp entre los time–stamp de los eventos que contiene o del último evento procesado si la cola está vacía.

Luego, cada Procesador Lógico sólo debe sacar un evento de la cola con *menor valor de reloj* entre todas las colas y procesar el evento. Como los Procesadores Lógicos se bloquean cuando una cola está vacía, puede ocurrir deadlock [1] si se genera un ciclo de espera entre los Procesadores Lógicos. Para subsanar este problema CMB utiliza mensajes nulos, mediante los cuales un Procesador Lógico notifica que no enviará más eventos. Cuando un Procesador Lógico recibe un mensaje nulo, puede avanzar su valor de reloj de la cola correspondiente y enviar posiblemente mensajes nulos a otros Procesadores Lógicos.

Como se ve, en CMB la restricción causal es respetada bloqueando los Procesadores Lógicos que no pueden avanzar su tiempo de simulación. Esta clase de algoritmo se llama *conservador* y su eficiencia no es óptima ya que no es mucho el cómputo que se realiza en paralelo. Existen varias opciones para mejorar la performance de CMB, entre ellas técnicas de *Lookahead*, la cual predice cuánto falta para que un Procesador Lógico reciba un evento desde otro y técnicas de *Análisis Estructural* para determinar las influencias entre los sub–modelos.

Sincronización Optimista – TimeWarp

Una segunda clase de algoritmos de sincronización es la llamada *optimista* en la cual los Procesadores Lógicos realizan la simulación tan rápido como puedan y si se encuentra una violación de la restricción causal se toman acciones para corregirla.

El algoritmo de *TimeWarp* fue presentado por Jefferson [47] y parecía ofrecer una solución a todos los problemas de la simulación en paralelo, aunque el costo asociado a algunos aspectos del algoritmo finalmente mostró lo contrario. TimeWarp permite que un Procesador Lógico avance su tiempo de simulación como si no hubiera necesidad de sincronización guardando los estados intermedios.

Si luego recibe un evento con menor time–stamp que su propio tiempo de simulación (un evento atrasado), el Procesador Lógico debe volver al estado guardado (rollback) exactamente previo al time–stamp del evento recibido y procesarlo. Obviamente el receptor debe a su vez deshacer los envíos a otros Procesadores Lógicos que haya realizado eso en ese período lo cual puede resultar en una cadena de rollbacks.

Simple y elegante como parece el algoritmo padece de algunos problemas. El consumo de memoria durante la simulación podría crecer sin límites. Por otro lado en algunos casos no se puede realizar rollback, por ejemplo cuando se realiza entrada/salida. Finalmente el costo computacional extra de guardar los estados intermedios y luego restaurarlos vuelve a TimeWarp en un algoritmo caro a la hora de implementar. Por esto, varias extensiones a la formulación original fueron realizadas. Un resumen de ellas puede ser encontrado en [57].

[1] Deadlock o abrazo mortal es una situación en la cual ningún procesador puede avanzar ya que todos esperan algún recurso que tiene otro.

NoTime

Como vimos previamente los algoritmos de sincronización y sus extensiones caen dentro de dos grandes categorías, optimistas y conservadores. Recientemente se desarrolló un algoritmo llamado NoTime [79] que no pertenece a ninguna de estas categorías.

La idea básica de NoTime es relajar la restricción causal. De esta forma permite a los Procesadores Lógicos avanzar su tiempo de simulación sin sincronizarlos. Los eventos atrasados son procesados como eventos correctos. Esto introduce cierto error en el resultado de la simulación ya que los tiempos de simulación variarán de un Procesador Lógico a otro. Aunque que esta técnica explota el paralelismo al máximo, puede dar resultados incorrectos o con error. De todas formas puede ser una técnica válida en situaciones en donde el usuario prefiere una simulación veloz frente a una solución precisa. El resultado de la simulación usando NoTime dependerá indirectamente de cómo está distribuido el trabajo entre los Procesadores Lógicos. Si éste no está balanceado algunos Procesadores Lógicos se ejecutarán rápidamente mientras que aquellos con más trabajo ejecutarán lentamente.

Capítulo 3

Simulación DEVS con PowerDEVS en Tiempo Real

Este Capítulo presenta el entorno de modelado y simulación PowerDEVS en su versión para tiempo real. Como dijimos antes, PowerDEVS fue extendido para correr en un sistema operativo de tiempo real (RTOS). Esta implementación se realizó utilizando el sistema RTAI [26].

El motor de simulación de PowerDEVS es una implementación en C++ del simulador descripto en la Sección 2.3.3. Como RTAI es una extensión de Linux, en principio las simulaciones de PowerDEVS que se ejecutan correctamente bajo Linux deberían correr también bajo RTAI.

Mientras que esto es cierto, el objetivo de ejecutar las simulaciones en RTAI es utilizar sus funciones para asegurar performance de tiempo real por lo cual varias modificaciones fueron hechas al motor de simulación de PowerDEVS. Estas modificaciones permiten a PowerDEVS sincronizar los eventos con el tiempo real, capturar interrupciones de hardware, utilizar archivos (desde tiempo real) y medir el tiempo de forma precisa.

La biblioteca de PowerDEVS fue también extendida con un conjunto de bloques que hacen uso de estas nuevas características y pueden ser utilizados arrastrándolos y conectándolos a los bloques comunes.

Este Capítulo introduce algunos conceptos relacionados a sistemas y simulación en tiempo real, describe las modificaciones al motor de simulación y los nuevos bloques de la biblioteca. Realizamos también algunos experimentos para medir la performance de PowerDEVS en tiempo real.

3.1. Motor de Simulación de PowerDEVS

Explicaremos en esta Sección la forma en que está implementado el algoritmo descripto en la Sección 2.3.3 dentro del programa que ejecuta la simulación.

Como PowerDEVS realiza la simulación utilizando un algoritmo escrito en C++, describiremos primero la estructura interna de clases.

Los modelos atómicos asociados al mismo código pertenecen a la misma clase. Por ejemplo, el modelo atómico *Integrator* en el modelo de la Figura 3.1 pertenece a la clase *Integrator* definida en los archivos *integrator.h* y *integrator.cpp* los cuales

contienen el código asociado al modelo atómico (transiciones interna, externa, función de avance de tiempo, etc).

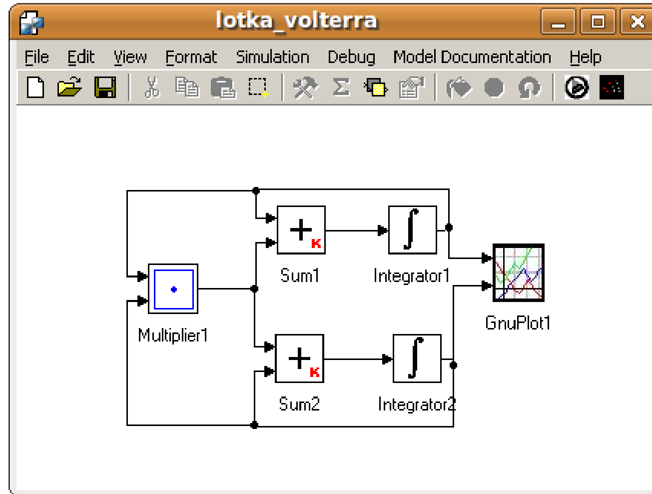

Figura 3.1: Ventana de Modelo PowerDEVS.

Todas las clases correspondientes a modelos atómicos heredan de la clase *simulator*. Esta clase es una clase abstracta que actúa como una interface para tratar con modelos atómicos. Las variables que representan el estado del modelo atómico serán variables miembro de la clase mientras que las funciones que operan sobre ellas (avance de tiempo, transiciones) serán métodos de la clase.

La función de transición externa y la función de salida de la clase *simulator* reciben y devuelven (respectivamente) objetos pertenecientes a la clase Event. Estos eventos poseen las siguientes propiedades:

- Event.port: es un entero que indica por qué puerto o hacia qué puerto el evento es recibido o enviado.
- Event.value: es un puntero a `void`, de esta forma los valores que contiene un evento pueden pertenecer a un tipo arbitrario.
- Event.realTimeMode: es un entero que puede tomar los valores: 0 (indicando que el evento no debe ser sincronizado con el tiempo real), 1 (utilizar sincronización normal) y 2 (utilizar sincronización precisa). Veremos en la Sección 3.2.1 que en el caso que el evento tenga modo distinto de cero el motor toma acciones para sincronizarlo con el tiempo real.

La estructura jerárquica de los modelos acoplados se implementa en la clase *coupling*. Esta clase es similar al coordinador de la Figura 2.7. Cada objeto de esta clase está asociado a un modelo DEVS acoplado y contiene una lista de conexiones internas, atómicos y acoplados hijos. Coherentemente con la propiedad de clausura de DEVS la clase *coupling* hereda de la clase *simulator*.

La clase *root–simulator* es la encargada de ejecutar la simulación. Básicamente esta clase va avanzando la simulación interactuando con el objeto que representa el acoplado raíz. Luego, el *root–simulator* juega el rol del root–coordinator en la Figura 2.7.

Finalmente, la clase principal de la simulación se llama *model*. Esta clase provee una consola interactiva entre el usuario y la simulación. Desde allí el usuario puede "hablar" directamente con el *root–simulator* permitiéndole iniciar, detener, o asignar distintos parámetros de la simulación.

Los objetos que forman la estructura jerárquica son instanciados en la inicialización. Esta función de inicialización es generada automáticamente por PowerDEVS ya que depende del modelo a simular. De hecho la única diferencia entre el código que ejecuta la simulación de distintos modelos es en esta función que instancia y inicializa los modelos DEVS y sus conexiones.

3.2. Implementación de PowerDEVS–RTAI

Presentaremos primero las modificaciones al algoritmo de simulación y luego cuatro nuevos módulos que utilizamos para su implementación en PowerDEVS. Estos módulos proveen funciones que pueden ser utilizados por cualquier modelo atómico.

3.2.1. Algoritmo de Simulación – Sincronización y Tiempo Real

Este módulo es la piedra fundamental para ejecutar PowerDEVS en tiempo real. Permite que el motor de simulación sincronice los eventos con el reloj de tiempo real de RTAI.

RTAI provee dos métodos para realizar una espera, ambos con precisión de ns:

- *rt_sleep*: detiene el proceso esperando una cantidad de tiempo liberando el procesador y dándoselo a otro hilo (si lo hubiera).
- *rt_busy_sleep*: detiene el proceso una cantidad de tiempo sin liberar el procesador en una espera busy–waiting, esto es, gasta ciclos de CPU hasta que el tiempo de espera finaliza. Con este método de espera se obtiene un resultado más preciso que con el anterior.

Basándose en estos métodos, el motor de simulación implementa la función `waitFor(unsigned long int nanoseconds, int mode)` la cual espera la cantidad de tiempo especificada en uno de dos modos: *Normal* o *Precise*. La función tiene la siguiente lógica:

- Si el tiempo de espera es menor que W_{min} la espera no se realiza.
- Si la función es invocada en modo *Normal*, se llama a la función sin busy–waiting *rt_sleep* (en la cual el CPU es liberado).
- Si la función es invocada en modo *Precise*, el comportamiento depende del tiempo de espera pedido:
    - Si el tiempo de espera es menor que W_{nobusy} se realiza una espera busy llamando a la función *rt_busy_sleep*.
    - En caso contrario, la espera se divide en dos sub–esperas. La primera espera tiene como longitud el período completo menos $2 * W_{nobusy}$, y ésta es realizada liberando el procesador llamando a *rt_sleep*.

> Al terminar este período, se mide nuevamente el tiempo (para eliminar el error introducido por la primera espera no–busy) y el tiempo faltante se espera con una espera busy. De esta forma la sincronización tiene la precisión de una espera busy y el procesador **es liberado la mayor parte del tiempo**.

Aquí, la constante W_{min} se define como la precisión de la espera busy en la plataforma elegida. Esta constante se utiliza para no realizar esperas en las cuales sólo la llamada a *rt_busy_sleep* consumiría más tiempo que la espera pedida. La segunda constante W_{nobusy} se define como la precisión de la espera no-busy.

La función `waitFor` fue definida así para lograr una espera precisa compartiendo el procesador la mayor cantidad del tiempo. Esta función puede ser invocada por cualquier modelo atómico. Es invocada automáticamente por el motor de simulación cuando un evento tiene su propiedad `realTimeMode` con un valor mayor a cero en la función de salida. Si un evento debe ser sincronizado debe ser creado en la función de salida con esta propiedad en un valor de 1 o 2. El motor esperará que el tiempo físico llegue al momento en el cual debe ser emitido el evento (si es que no pasó ya), y luego propagará el evento y ejecutará la transición interna del modelo que emitió el evento.

De esta forma, dentro del mismo modelo, algunos eventos pueden ser propagados inmediatamente (cuando tienen su propiedad `realTimeMode` igual a cero) y otros pueden ser sincronizados tanto en forma precisa como normal. Esto da al usuario la posibilidad de elegir cuáles eventos deben ser sincronizados (típicamente aquellos involucrados en la comunicación con el mundo) y cuáles deben ser calculados y propagados lo más rápido posible (aquellos involucrados en cálculos intermedios).

3.2.2. Manejo de Interrupciones y Archivos

Este módulo es el encargado de la interacción entre el motor de simulación de PowerDEVS y las interrupciones de hardware

Un modelo atómico puede declarar su interés en una interrupción de hardware llamando a la función[1]:

```
requestIRQ(int irq);
```

Cuando la interrupción `irq` ocurre, el motor de simulación genera una transición externa en todos los modelos atómicos (en orden de prioridad) que han expresado interés en esa interrupción. El evento se envía a un puerto inexistente (denotado como -1). Esto es para poder diferenciarlos de eventos provenientes de otros modelos atómicos.

Es posible que al recibir esta transición externa el modelo atómico receptor cambie su tiempo de avance $ta(s)$. El motor de simulación transmite este cambio (si lo hubiera) hacia arriba en la estructura jerárquica de atómicos–acoplados. Este es el mecanismo que mencionamos en la Sección 2.3.3 como una modificación al simulador abstracto de [94].

Cuando un proceso está corriendo bajo RTAI, no puede hacer uso de los servicios provistos por Linux (como el Sistema de Archivos). Esto se debe a que una llamada al sistema al núcleo de Linux no de una cota en el tiempo de respuesta

[1] En la arquitectura i386 las interrupciones pueden traducirse en un entero

y podría violarse un deadline mientras esta llamada es ejecutada. De hecho las tareas de RTAI no son manejadas por el scheduler de Linux el cual no sabe de la existencia de RTAI. Por lo tanto para permitir el uso de archivos desde una simulación desarrollamos el módulo *Soporte de Archivos*.

Este módulo contiene dos partes:

La interfaz de tiempo real: Es una tarea de tiempo real (corriendo bajo RTAI) que acepta pedidos de la simulación (como abrir, leer, escribir archivos). Esta tarea se comunica con un programa corriendo bajo Linux (en espacio de usuario) que tiene acceso al sistema de archivos de Linux. Esta comunicación se realiza utilizando un FIFO de tiempo real (uno de los mecanismo de IPC provistos por RTAI).

Su contra-parte en espacio de usuario: Es un programa normal de Linux que acepta comandos de la interfaz de tiempo real. Este programa (que es lanzado junto con la simulación) recibe los pedidos a través del FIFO y realiza las llamadas en nombre de la simulación.

La interfaz provee funciones similares a aquellas de la biblioteca Standard C (stdio) que puede ser utilizada por cualquier modelo atómico.

3.2.3. Medición del Tiempo

Este módulo permite a los modelos atómicos utilizar el reloj de tiempo real de RTAI para medir el tiempo físico o *wall–clock*. Esto es útil por ejemplo cuando un usuario de PowerDEVS quiere cambiar el comportamiento del modelo dependiendo de la diferencia entre el tiempo real y el tiempo de simulación. Por ejemplo si se estuviera en una situación de overrun [1] podría bajar (dinámicamente) la precisión de la simulación; o aumentarla si detecta que hay tiempo suficiente para una simulación más precisa.

3.2.4. Desempeño en Tiempo–Real

Una de las características más importantes para caracterizar un sistema de tiempo real es el *error de sincronización* o latencia. Realizamos varios experimentos para ver cómo el sistema responde (en función de la carga del sistema). Todos los experimentos de este Capítulo se realizaron en una PC AMD Athlon 1.8 GHz utilizando RTAI 3.6, Linux 2.6.31 y PowerDEVS 2.2.

Primero, realizamos el siguiente experimento: simulamos un modelo que consiste en una fuente senoidal (aproximada por QSS) con frecuencia de 440Hz (el modelo genera 20 eventos por ciclo, para un total de 8800 por segundo). El modelo fue simulado en tiempo real midiendo la latencia máxima y promedio. Los resultados pueden verse en el Cuadro 3.1. Vemos que con poca carga computacional no hay overrun, lo cual es deseable. También ambos modos de espera fueron probados (ver Sección 3.2.1) obteniendo los siguientes resultados:

Luego, simulamos un modelo más complejo: un control Proporcional Integral (PI) de un motor de corriente continua (DC) usando Pulse Width Modulation

[1] overrun significa que el deadline para los cómputos ya se ha vencido

Tabla 3.1: Error de sincronización con poca carga computacional.

Espera	Máxima	Promedio	Overruns
Normal	4800 ns	1500 ns	-
Precise	450 ns	180 ns	-

(PWM). Ejecutamos varias simulaciones variando la frecuencia de la portadora[1] de PWM desde 1000 hasta 20000 Hz:

Tabla 3.2: Error de sincronización variando la carga.

f (Hz)	Modo de Espera	Máxima	Promedio .	Overruns(eventos)
1000	Normal	5786 ns	1512 ns	-
	Precise	1330 ns	180 ns	-
15000	Normal	5622 ns	1622 ns	372 / 19905 ev.
	Precise	1000 ns	512 ns	305 / 19905 ev.
17000	Normal	4547 ns	1648 ns	6119 / 17616 ev.
	Precise	973 ns	454 ns	5924 / 17616 ev.
20000	Normal	0 ns	0 ns	18292 / 18292 ev.
	Precise	0 ns	0 ns	18292 / 18292 ev.

Como puede verse en la Tabla 3.2 utilizando una frecuencia de 1000 Hz no hay *overruns*. Con frecuencias entre 15000 Hz y 17000 Hz hay más *overruns* y el sistema Linux (no el sistema de tiempo real) experimenta algunas demoras lo que indica que el sistema tiene una carga computacional alta. Con una frecuencia de 20000 Hz el sistema Linux deja de responder (es decir la tarea de tiempo real no libera el procesador para que lo use Linux) y vemos que todos los eventos son emitidos **después** de su deadline (en overrun). Este último caso refleja un límite de la plataforma de hardware, no una limitación de la sincronización del sistema. Lo que ocurre con una frecuencia de 20000Hz es que el hardware no puede completar los cálculos a tiempo.

3.2.5. Latencia de Interrupción

Otra características importante a la hora de implementar un sistema de tiempo real es qué tan rápido responde a estímulos externos. Para ello medimos la latencia con la cual el sistema responde a las interrupciones de hardware.

Para medir la latencia de interrupción utilizamos el siguiente procedimiento. Creamos un modelo de PowerDEVS (Figura 3.2) que genera interrupciones de hardware y luego captura nuevamente estas interrupciones midiendo el tiempo que transcurre entre la generación de la señal que provoca la interrupción y la captura de esta.

Para este experimento utilizamos el puerto paralelo de la PC. Uno de los pins del puerto paralelo (STO) es el encargado de generar las interrupciones. Este pin

[1] La señal portadora de PWM es en general una señal diente de sierra

fue conectado a un pin de datos, generando por lo tanto una interrupción cada vez que el pin de datos va de un estado bajo (0V) a un estado alto (5V).

El modelo guarda el tiempo al cual el 1 es escrito al pin de datos (llevándolo al estado alto) y el tiempo al cual el manejador de interrupción del puerto paralelo es ejecutado. La diferencia entre estos dos tiempos es un límite **superior** en la latencia de interrupción del sistema (de hecho esta diferencia incluye también el tiempo necesario para escribir al puerto paralelo y los transitorios eléctricos).

Ejecutando este experimento en la plataforma de hardware antes descripta obtuvimos un límite superior en la latencia de interrupción cercano a los 20μs lo cual es más que suficiente para la mayoría de las aplicaciones de tiempo real.

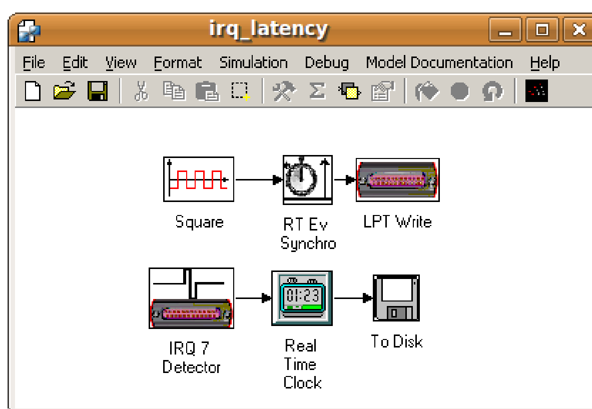

Figura 3.2: Modelo para medir la latencia de interrupción.

3.3. Biblioteca de Modelos para Tiempo–Real

Para hacer uso de las características de tiempo real de una forma simple la biblioteca de PowerDEVS fue extendida para incluir varios bloques que pueden ser incorporados directamente en los modelos.

RTWait: Este bloque atómico recibe eventos y los emite sincronizados con el reloj de tiempo real. De esta forma un modelo normal (un modelo sin tiempo real) puede ser transformado en un modelo de tiempo real insertando instancias de estos bloques en las conexiones que transmiten eventos que necesitan ser sincronizados. El bloque tiene un parámetro para elegir que tipo de sincronización debe ser realizada, *precise* o *normal* (ver 3.2.1).

RTClock: Cuando este bloque recibe un evento (sin importar el valor del evento) emite un evento con valor igual al tiempo medido con el reloj del tiempo real.

ToDisk: Este bloque escribe su señal de entrada en un archivo CSV(Comma Separated Values). El bloque hace uso del módulo de Soporte de Archivos en tiempo real.

IRQDetector: Este modelo es la interfaz de usuario para el módulo de Manejo de Interrupciones. El bloque emite un evento cuando ocurre una interrupción de hardware. Desde el punto de vista del formalismo DEVS (y desde PowerDEVS también) este bloque es totalmente pasivo (su tiempo de avance es ∞), pero cuando la interrupción asociada es disparada, el modelo cambia su propio tiempo de avance a 0 (y el motor de simulación automáticamente notifica este cambio a su padre). El bloque tiene un parámetro para indicar por cual interrupción esperar.

3.4. Comunicación con Scilab

Scilab [40] es un paquete computacional desarrollado por el Institut National de Recherche en Informatique et Automatique (INRIA). Es liberado bajo licencia GPL. Scilab es una de las alternativas de código abierto para el software Matlab.

Scilab contiene una interfaz interactiva donde el usuario puede definir variables, matrices y realizar operaciones complejas entre ellos. También posee un lenguaje de programación que puede ser utilizado para definir nuevas funciones y algoritmos y herramientas gráficas para mostrar datos. La distribución de Scilab incluye varios toolboxes (un conjunto de funciones y algoritmos apuntados a un fin común) para resolver problemas de álgebra lineal, procesamiento de señales, control automático, optimización diseño de filtros, etc.

Desarrollamos junto con la versión de tiempo real de PowerDEVS una interfaz para comunicar la herramienta con Scilab [27]. Para realizar la comunicación entre las dos herramientas desarrollamos un módulo de Scilab llamado **backdoor** el cual "abre" una *puerta trasera* al entorno de Scilab, mediante la cual las variables de Scilab pueden ser leídas y escritas desde PowerDEVS. La comunicación se realiza a través de una conexión TCP sobre el puerto 27015 bajo una arquitectura cliente-servidor (donde Scilab es el servidor y PowerDEVS el cliente).

El Lado de Scilab

En el trabajo original [27] se investigó una primera versión de la interfaz Scilab–PowerDEVS fue investigada. En ella se modificó el código de Scilab 4.1.2 (ya que es OpenSource) para llevar a cabo la comunicación. Modificar el código fuente implica que el entorno completo de Scilab debe ser recompilado para cada una de las arquitecturas (Windows/Linux) y el archivo binario modificado debe ser distribuido con PowerDEVS.

Durante el transcurso de esta Tesis se actualizó la interfaz para realizarla de forma más limpia y elegante. Desde su versión 5.2, Scilab ofrece la posibilidad de extenderlo mediante módulos escritos por los usuarios. Desarrollamos nueva versión de la interfaz Scilab–PowerDEVS como módulo de Scilab, de esta forma, no debemos compilar Scilab sino que sólo distribuimos el módulo y Scilab se encarga de compilarlo y cargarlo.

Este módulo llamado **BackDoor** [7] crea un nuevo hilo de ejecución dentro de Scilab de forma de que la interfaz gráfica siga respondiendo a los comandos del usuario mientras que el nuevo hilo es el encargado de recibir, ejecutar y responder los pedidos de PowerDEVS. Este hilo (de Scilab) es el servidor al cual se comunica el motor de simulación de PowerDEVS (ver Figura 3.3).

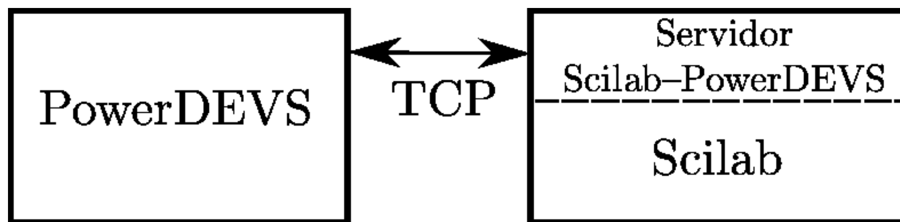

Figura 3.3: Comunicación Scilab–PowerDEVS.

Los pedidos de PowerDEVS enviados a Scilab consisten de una representación en cadena de caracteres del comando que debe ser ejecutado en Scilab (por ejemplo “freq=pi*345”). Cada pedido incluye información acerca del modo del pedido, este puede ser:

Bloqueante El motor de simulación de PowerDEVS envía el pedido y espera a que Scilab le devuelva un resultado.

No-Bloqueante El motor no espera una respuesta de Scilab. Este modo no puede ser utilizado si los pedidos sucesivos dependen de la ejecución del pedido actual.

La interfaz Scilab–PowerDEVS puede recibir también un pedido especial, en el cual no ejecuta ninguna sentencia Scilab, sino que sólo devuelve el resultado de la variable “ans” (el resultado del último cálculo). Esto permite –indirectamente– evaluar cualquier expresión o variable de Scilab desde PowerDEVS enviando una secuencia de comandos correctos.

El Lado de PowerDEVS

El motor de simulación de PowerDEVS contiene un conjunto de servicios para comunicarse con Scilab. Éstos puede ser invocados desde cualquier modelo atómico:

```
void  getAns(double *ans, int rows, int cols);
void  putScilabVar(char *varname, double v);
void  getScilabMatrix(char *varname, int *rows, int *cols, double **data);
void  getScilabVector(char *varname, int *length, double *data);
double  getScilabVar(char *varname);
double  executeScilabJob(char *job, bool blocking);
double  executeVoidScilabJob(char *job,bool blocking);
```

donde cada función realiza lo siguiente:

getAns devuelve el resultado del último cálculo realizado en el entorno de Scilab (esto es la variable “ans”). El resultado es escrito en el valor de punto flotante apuntado por `ans`. Los argumentos `rows` and `cols` indican el tamaño del resultado que debe ser recuperado (Scilab trabaja con matrices por lo cual también lo hace la interfaz con PowerDEVS). Si el resultado es un valor escalar `r` y `c` deben ser `1`.

putScilabVar y getScilabVar son los métodos básicos para interactuar con variables de Scilab. La primera crea (o actualiza) una variable de Scilab llamada `varname` con el valor `v`. La segunda lee el valor de la variable de Scilab `varname`. Si la variable no existe la función devuelve `0.0` y escribe un mensaje de advertencia en el log de la simulación.

getScilabMatrix y getScilabVector son versiones matriciales y vectoriales de la función getScilabVar aunque éstas no requieren el tamaño de la variable como argumento sino que toman el tamaño del entorno de Scilab y lo escribe en lo apuntado por las variables `rows`, `cols` y `length` y escribe los datos en lo apuntado por `data`. Este último puntero debe apuntar a un lugar de memoria suficiente para alocar `sizeof(double)*length` o `sizeof(double)*rows*cols` correspondientemente.

executeScilabJob y executeVoidScilabJob ejecutan un comando (`job`) en el entorno de Scilab. El resultado es el mismo que si uno escribiera la sentencia en la ventana de comando de Scilab. El segundo argumento indica si la llamada debe ser realizada de modo bloqueante o no. La diferencia entre las dos es que la segunda ejecuta el comando **sin sobreescribir** el valor de la variable “ans” lo que es necesario en algunos casos.

Parámetros desde Scilab y Bloques de Interfaz

Junto con el desarrollo de la interfaz Scilab–PowerDEVS realizamos varias modificaciones a la biblioteca de PowerDEVS para simplificarle al usuario el acceso a Scilab.

Primero se modificó el código de todos los bloques atómicos de la biblioteca para que acepten expresiones de Scilab como valores en sus parámetros. Por ejemplo, uno puede usar “*freq*2+0.5*” como parámetro a un bloque y esta expresión es evaluada en el entorno de Scilab (en este caso *freq* debe ser una variable ya definida). Como veremos más adelante en el Capítulo 4 los bloques vectoriales también son capaces de tomar sus parámetros desde una variable o matriz de Scilab.

También se agregaron tres bloques a la biblioteca de PowerDEVS:

ToWorkspace Este bloque exporta la señal continua de entrada a dos variables de Scilab (cuyo nombre toma como parámetro). Cómo los eventos pueden no estar equi–espaciados en el tiempo la señal se escribe en una variable que contiene el tiempo del evento y otra el valor. El tamaño de los arreglos generados en el entorno de Scilab dependerá de la cantidad de eventos recibidos por el bloque durante la simulación.

FromWorkspace Este bloque es análogo al anterior, toma una señal del entorno de Scilab descripta por dos vectores (tiempo y valor) y emite esos eventos en PowerDEVS hasta que finalice la simulación o se acabe el vector de origen. Cabe destacar que este bloque opcionalmente interpola los valores utilizando QSS2 o QSS3.

ScilabCommand Este bloque ejecuta comandos (que toma como parámetros) en el entorno de Scilab. Puede ejecutar comandos al inicializar el bloque, cada vez que recibe un evento por su puerto de entrada, y/o en el final de la simulación.

3.5. Ejemplos y Resultados

En esta Sección mostramos dos ejemplos que muestran distintas características de PowerDEVS en tiempo real y de la comunicación con Scilab.

3.5.1. Ripple vs Frecuencia en una Fuente Conmutada Buck

La Figura 3.4 muestra un circuito convertidor DC-DC conocido como *Fuente Conmutada Buck*.

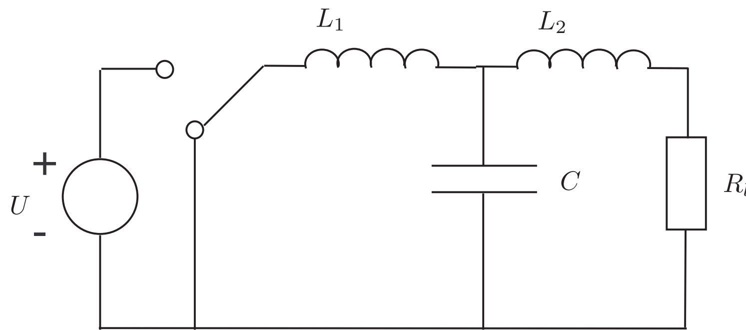

Figura 3.4: Circuito Fuente Conmutada Buck.

La presencia de la llave introduce un comportamiento híbrido al sistema.

El objetivo de este experimento es analizar la dependencia de la frecuencia de conmutación en la amplitud de "ripple". En este ejemplo utilizamos los siguientes parámetros $L_1 = L_2 = 0{,}1$mH, $C = 20\mu$F, $R_l = 10\Omega$ y $U = 12$V.

La Figura 3.5 muestra el modelo PowerDEVS del circuito realizado completamente con bloques de la biblioteca de PowerDEVS. La llave está comandada por una señal PWM. Todos los bloques (fuentes, funciones estáticas, integradores) están basados en los métodos de QSS.

La frecuencia de la portadora triangular fue elegida de forma que se incremente en 2000Hz en cada simulación.

El bloque "RunScilab Job" incrementa la variable de Scilab `n` al finalizar cada simulación y calcula la amplitud del ripple en régimen estacionario. Por esta razón el bloque está desconectado del resto del sistema. Esta amplitud es escrita en el arreglo `u(n)` de Scilab.

Luego de 100 simulaciones (en la cual la frecuencia va desde 2000 hasta 200000 Hz), podemos graficar los resultados directamente en Scilab (Figura 3.6).

Notemos aquí que 100 simulaciones de este sistema híbrido (con conmutaciones rápidas) fueron realizadas y gracias a la eficiencia de los métodos de QSS para tratar con este tipo de problemas, los experimentos sólo tomaron 13 segundos (mientras que toma minutos en herramientas como Matlab/Simulink).

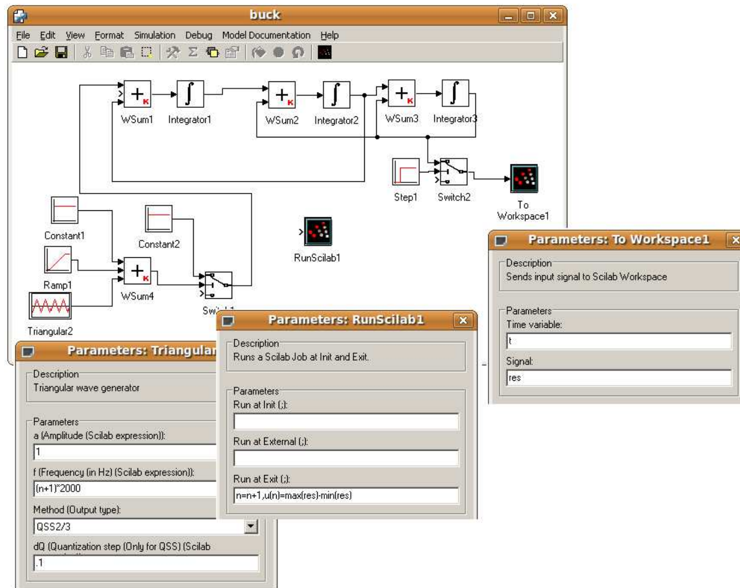

Figura 3.5: Modelo circuito Fuente Conmutada Buck.

3.5.2. Control de un Motor de Corriente Continua en Tiempo–Real

Este ejemplo muestra el control asíncrono de un motor DC. Para este fin utilizamos un pequeño motor DC moviendo una rueda de mouse que hace de codificador rotatorio (o encoder) (ver Figura 3.7).

Cada vez que la rueda se mueve un pequeño ángulo, un pulso es enviado al bit *STO* del puerto paralelo, el cual genera una interrupción de hardware en la PC. El motor es alimentado por un amplificador implementado con un transistor y un filtro tomando el voltaje on–off de un pin de datos del puerto paralelo.

El sistema de control está compuesto de los siguientes subsistemas (ver Figura 3.8):

- El bloque *Motor Speed* que recibe y cuenta las interrupciones generadas por en encoder para estimar la velocidad del motor.
- Esta estima es comparada a la velocidad de referencia (el bloque *WSum4* calcula la diferencia o error).
- En base a este error un control Proporcional–Integral (PI) es aplicado utilizando la discretización de QSS.
- La señal de control (previamente saturada para evitar sobre-modulación) es modulada con PWM.
- La señal PWM (calculada en el bloque *Comparator1*), es enviada *sincronizada con el tiempo real* al bit de datos del puerto paralelo.

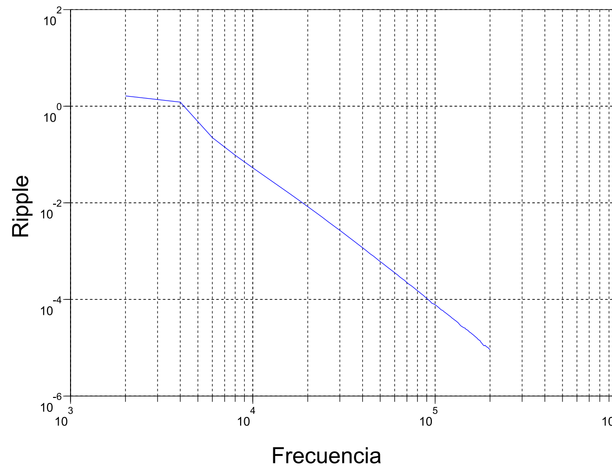

Figura 3.6: Ripple vs. Frecuencia.

Todo el sistema de control de la Figura 3.8 fue creado utilizando bloques de la biblioteca de PowerDEVS excepto por el modelo que estima la velocidad del motor en función del número de interrupciones que recibe por segundo que fue realizado creando bloques específicos para este ejemplo.

La Figura 3.9 muestra la velocidad de referencia mientras que la Figura 3.10 muestra la velocidad medida por el sistema de control. En ellas vemos que el control mantiene la velocidad cercana a la velocidad de referencia. Este sistema no podría ser ejecutado fuera de tiempo real, sin una sincronización precisa.

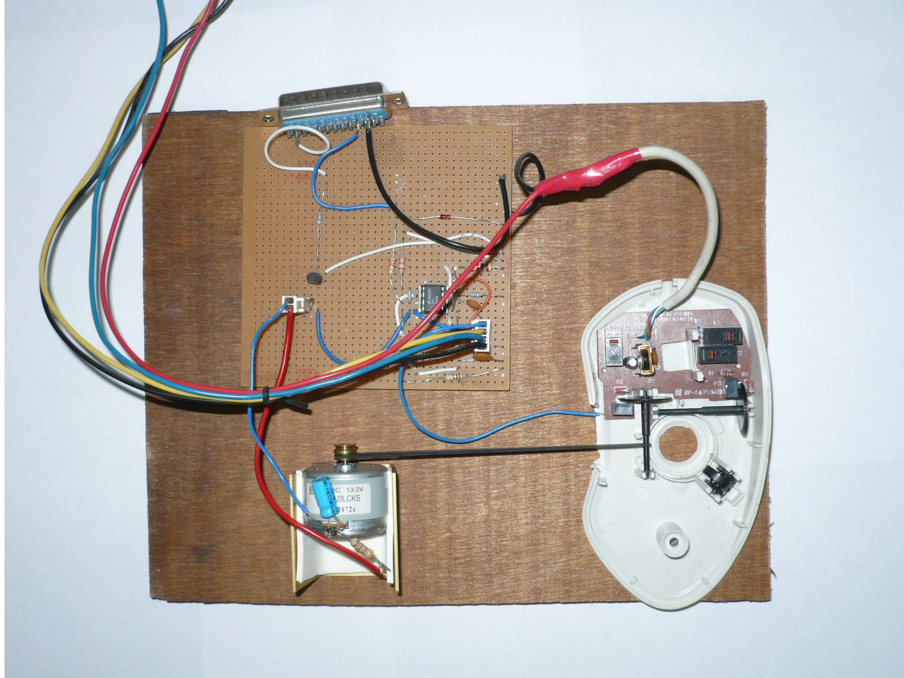

Figura 3.7: Motor y mouse (encoder).

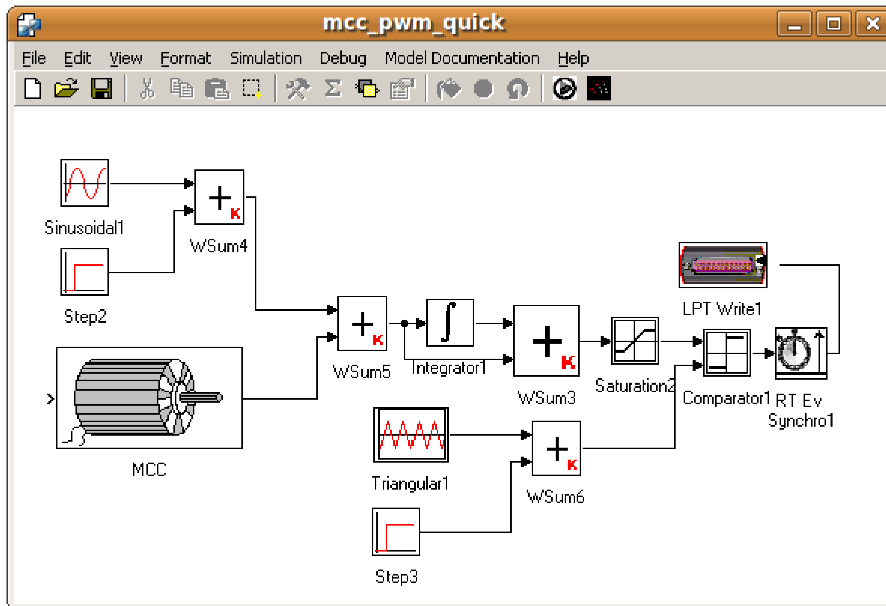

Figura 3.8: Modelo del Control Proporcional.

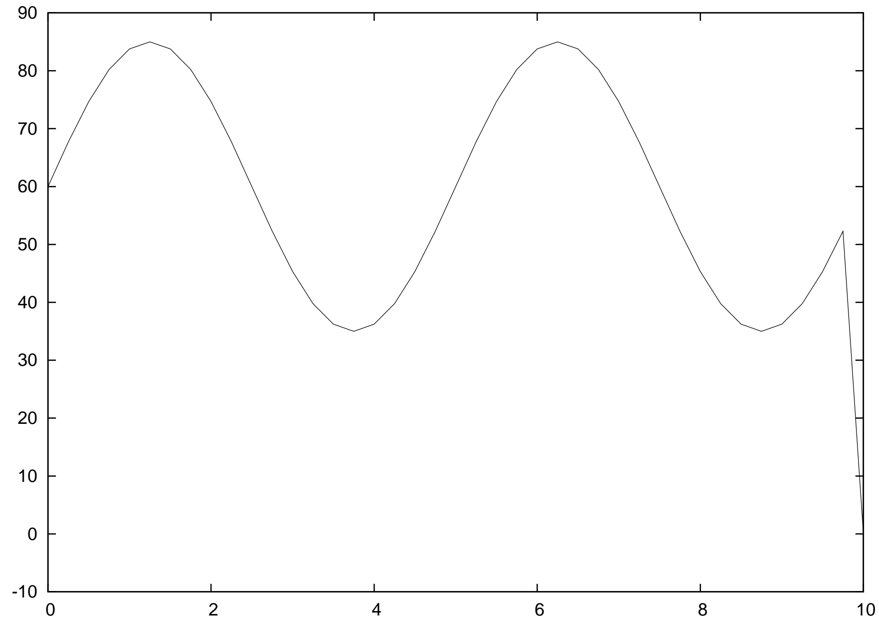

Figura 3.9: Velocidad de referencia.

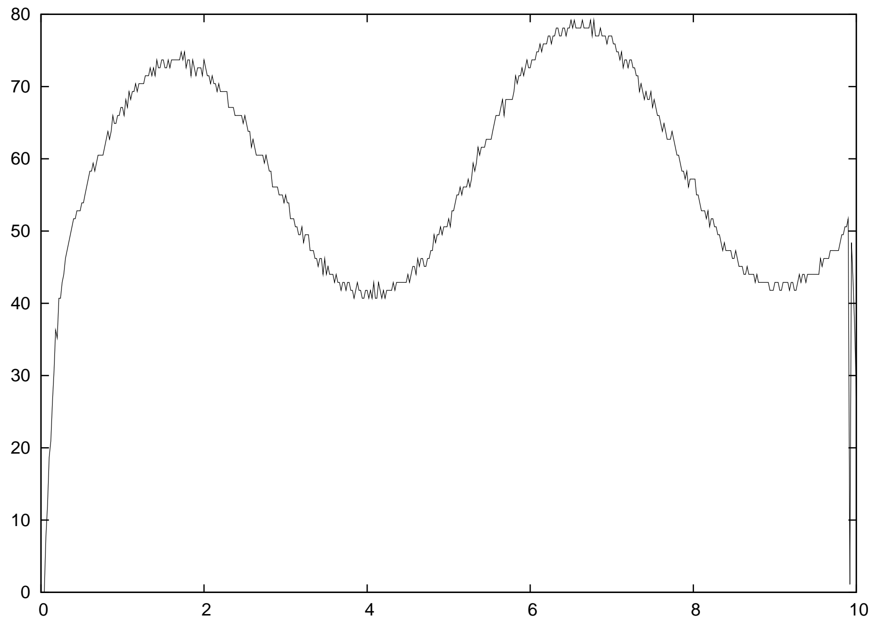

Figura 3.10: Velocidad medida.

3.6. Conclusiones

Introducimos en este capítulo una herramienta para la simulación de tiempo real basada en DEVS, llamada RTAI–PowerDEVS. Esta herramienta permite ejecutar las simulaciones sincronizado el tiempo lógico con el tiempo real con la posibilidad de manejar interrupciones de forma sencilla. De esta manera controladores híbridos basados en QSS pueden ser desarrollados directamente sin mucha programación (como lo muestra el ejemplo del control de motor DC).

Esta plataforma ha sido utilizada en varios trabajos como base para desarrollar sistemas MIL y HIL [3, 5, 37].

Veremos en el Capítulo 5 el uso de este entorno para la simulación en paralelo de modelos DEVS. Allí se utiliza la sincronización de los eventos emitidos como forma de coordinar las distintas unidades de procesamiento.

Capítulo 4

Modelos DEVS Vectoriales

En este Capítulo introducimos una extensión al formalismo DEVS llamada Vectorial DEVS (VECDEVS) que permite representar sistemas grandes gráficamente mediante diagrama de bloques. Un modelo VECDEVS puro consiste en un arreglo de modelos DEVS clásicos idénticos que pueden diferir en sus parámetros.

La interconexión de modelos VECDEVS puede ser realizada a través de algunos modelos DEVS clásicos especiales permitiendo representar fácilmente sistemas grandes de estructura arbitraria.

Una característica importante de esta extensión es que los modelos VECDEVS pueden ser divididos fácilmente para su simulación en paralelo. Para ese fin, hemos desarrollado un algoritmo que particiona un modelo VECDEVS automáticamente en un número arbitrario de submodelos para su simulación en paralelo.

Describimos también en este Capítulo la implementación de VECDEVS en la herramienta PowerDEVS, el algoritmo de particionado y mostramos su uso a través de algunos ejemplos.

4.1. Introducción

La mayoría de las herramientas de modelado proveen una interfaz gráfica que permite crear modelos y conectar diferentes subsistemas. El acoplamiento jerárquico modular de subsistemas es la forma más común de componer sistemas en casi todos los dominios técnicos.

Sin embargo, en presencia de grandes sistemas, el acoplamiento jerárquico modular gráfico no es suficiente ya que se vuelve dificultoso (o imposible) crear y conectar miles de modelos de forma gráfica. Para este propósito, los diferentes lenguajes de modelado permiten el uso de modelos vectoriales, donde componentes con una representación gráfica simple pueden contener muchas instancias idénticas de subsistemas elementales.

DEVS [94] es el formalismo más general para modelar sistemas de eventos discretos. Permite representar cualquier sistema que realice un número finito de cambios en un intervalo finito de tiempo. No sólo las Redes de Petri [69], State–Charts [41] Event–Graphs [81] y otros formalismos de eventos discretos pueden ser representados en DEVS sino que también pueden pensarse los sistemas de tiempo discreto como casos particulares de DEVS [87, 93].

La principal extensión DEVS para soportar modelado y simulación de grandes sistemas es Cell-DEVS [91]. Sin embargo, los modelos celulares no pueden representar fácilmente modelos grandes con estructura compleja y una extensión vectorial al formalismo DEVS parece una herramienta necesaria.

Motivados por esta necesidad, proponemos el formalismo Vectorial DEVS el cual provee una forma de representar modelos grandes de una manera sencilla.

Un modelo atómico VECDEVS consiste en un arreglo de modelos DEVS atómicos idénticos (que pueden diferir en sus parámetros). Éstos pueden ser acoplados en una jerarquía de modelos, lo cual resulta equivalente a un arreglo de modelos DEVS acoplados idénticos desconectados. Luego, la técnica VECDEVS se completa con algunos modelos DEVS atómicos que pueden manejar eventos VECDEVS para provocar conexiones entre diferentes copias y también conectar modelos VECDEVS con modelos DEVS regulares. De este modo, el nuevo formalismo permite representar sistemas grandes de estructura compleja.

La simulación de sistemas grandes requiere generalmente el uso de técnicas de paralelización. Una de las principales ventajas de los modelos VECDEVS es que pueden ser divididos fácilmente para su simulación en paralelo. Hemos desarrollado un algoritmo para el particionado automático de modelos Vectorial DEVS. Este algoritmo puede ser utilizado con las técnicas que veremos en el Capítulo 5.

Hemos desarrollado también una biblioteca e implementamos el algoritmo de particionado en PowerDEVS.

El Capítulo presenta primero algunos formalismos y herramientas relacionadas (Sección 4.2) y luego en la Sección 4.3 presentamos la extensión Vectorial DEVS. En la Sección 4.4 discutimos la implementación de Vectorial DEVS en PowerDEVS, después presentamos el algoritmo de particionado (Sección 4.5) y finalmente presentamos varios casos de uso en la Sección 4.6 y algunas conclusiones en la Sección 4.7.

4.2. Otros Formalismos y Herramientas

Existen en la literatura tanto formalismos como herramientas de software que soportan el modelado vectorial.

Modelos DEVS de gran escala

Dentro del mundo de DEVS existen algunos enfoques similares al propuesto en este Capítulo. Cell–DEVS [91] es un autómata celular (CA) especializado. Un CA es un retículo regular n–dimensional donde cada una de las celdas puede tomar un valor finito. Los estados en el retículo son actualizados de acuerdo a un conjunto de reglas de forma síncrona y todas las celdas cambian su estado en pasos de tiempo discretos.

Cell–DEVS es un combinación de DEVS y autómatas celulares con retardo temporales explícitos. En Cell–DEVS cada modelo se define como un modelo atómico de DEVS y se define un procedimiento para acoplar cada celda. Mientras que este formalismo es similar al propuesto en el presente trabajo, tiene desventajas a la hora de modelar sistemas con conexiones irregulares.

Arreglo de Modelos

Varias herramientas y lenguajes soportan la definición de arreglo o vectores de modelos. De esta forma, varias copias de un mismo modelo puede ser definido e interconectado produciendo modelos más complejos que los definidos a través de de Cell–DEVS. Entre las diferentes herramientas y lenguajes destacamos:

- Statechart [41] es un lenguaje de modelado gráfico. Es una extensión de las máquinas de estado y diagramas de estado. Es utilizado para especificar y diseñar sistemas de eventos discretos complejos. El usuario puede describir modelos vectoriales, por ejemplo para modelar N instancias de un modelo M uno puede dibujar un cuadro tridimensional donde cada capa de este cuadro es un modelo de clase M (ver Figura 4.1). Aquí también la conexión entre estos N modelos está muy restringida. Todos los modelos se conectan bajo una estructura regular.

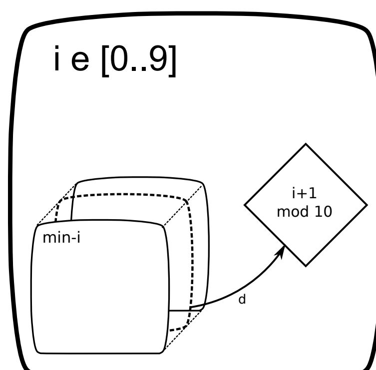

Figura 4.1: Representación gráfica de un StateChart vectorial.

- Modelica [34, 35] es un lenguaje orientado a objetos basado en ecuaciones que permite representar tanto sistemas continuos como híbridos usando un conjunto de ecuaciones no-causales. El lenguaje Modelica provee una forma estandarizada para modelar sistemas físicos complejos multidominio que contienen por ejemplo subcomponentes mecánicos, eléctricos, electrónicos, hidráulicos, térmicos, de control, de energía eléctrica u orientados a procesos. Al ser un lenguaje orientado a objetos el usuario de Modelica puede definir arreglos de modelos y conectarlos programáticamente. Existen también herramientas comerciales como Dymola [74] junto con implementaciones de código abierto como Scicos [70] y OpenModelica [33].

  Estas herramientas poseen una interfaz gráfica donde uno puede conectar distintos submodelos que luego la herramienta traducirá a conexiones en el lenguaje Modelica. El modelo es luego convertido a una descripción C que es compilada junto con el integrador y ejecutada para realizar la simulación.

  En la versión actual de OpenModelica (1.7) los modelos vectoriales no están implementados como arreglos de C lo cual degrada la performance ya que crea tantas variables como el tamaño del arreglo. Se está trabajando en OpenModelica para no expandir los arreglos pero esto todavía no está completo. Como veremos en la Sección 4.4 la implementación de Vectorial DEVS

en PowerDEVS sí usa arreglos de C. Dymola es una herramienta comercial (de código propietario) por lo cual no sabemos cómo están implementados los arreglos.

- El sistema Simulink [50] de Mathworks es un entorno comercial de simulación multidominio basado en el diseño de modelos para sistemas dinámicos y embebidos. Provee una atractiva interfaz gráfica junto con una gran biblioteca de bloques que permiten el diseño, simulación, implementación y verificación de una variedad de sistemas dinámicos. Simulink soporta también modelado vectorial, posee el concepto de un *Bus* donde múltiples señales pueden ser transmitidas por el mismo "cable". Cuando un *Bus* es conectado a un bloque de la biblioteca (por ejemplo un integrador) este bloque es replicado tantas veces como el tamaño del bus obteniendo así una versión vectorial del bloque. Simulink tiene también bloques para multiplexar y demultiplexar señales escalares hacia el bus y vice-versa y bloques para seleccionar un subconjunto de las señales del bus. Esto permite al usuario de Simulink la posibilidad de modelar sistemas de gran escala de forma gráfica. A diferencia de PowerDEVS, Simulink no está basado en DEVS sino que está basado en ecuaciones diferenciales y diagrama en bloques. Al ser Simulink una herramienta comercial no podemos especificar cómo estas características están implementados.

4.3. Modelos Vectorial DEVS

En esta Sección introducimos una extensión al formalismo DEVS para especificar modelos vectoriales fácilmente. Primero discutimos un ejemplo motivador para mostrar la necesidad de este nuevo formalismo. Luego introducimos modelos DEVS parametrizados, que constituyen la base de VECDEVS y después presentamos el formalismo Vectorial DEVS.

4.3.1. Ejemplo Motivador

El siguiente ejemplo fue tomado de [75]. Allí se presenta un modelo para estudiar el consumo energético de un conjunto de Aires Acondicionados (AA).

La idea es que cada AA controla la temperatura de una habitación distinta. Cada habitación tiene una temperatura distinta, resistencia térmica, y capacitancia térmica, y sufre de distintas perturbaciones.

Cada AA mantiene la temperatura de la habitación cercana a una temperatura de referencia común, prendiendo y apagando el sistema de enfriado. Para ello, cada AA sigue una ley de prendido-apagado con histéresis.

La evolución de la temperatura de una habitación $\theta(t)$ se describe por una ecuación diferencial:

$$\frac{d\theta(t)}{dt} = -\frac{1}{C \cdot R}[\theta(t) - \theta_a + R \cdot P \cdot m(t) + w(t)], \tag{4.1}$$

donde R y C son la resistencia y capacitancia térmica, respectivamente. P es la potencia del AA cuando está prendido, θ_a es la temperatura ambiente externa y $w(t)$ es un término que representa las perturbaciones térmicas.

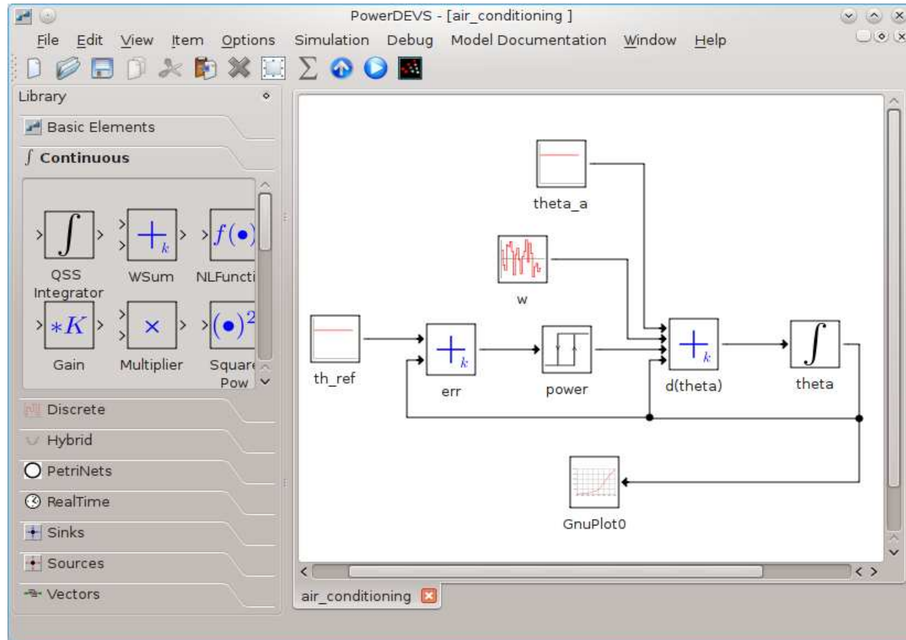

Figura 4.2: PowerDEVS Model Window.

La Figura 4.2 muestra el modelo de un AA en PowerDEVS. Allí cada uno de los bloques son modelos DEVS atómicos resultantes de utilizar los métodos de QSS para aproximar la ecuación 4.1.

El modelo utilizado en [75], está compuesto de 10.000 AAs, donde *cada* habitación tiene diferentes parámetros R y C, recibiendo distintas perturbaciones $w(t)$ y donde cada AA tiene un consumo P distinto. El consumo total es el resultado de sumar los consumos individuales de cada AA (la potencia es la salida del bloque de histéresis etiquetado 'power' en la Figura 4.2).

Luego, si queremos construir el modelo completo, debemos replicar el modelo de la Figura 4.2 10.000 veces, cambiando los parámetros y luego deberíamos crear un bloque sumador con 10.000 entradas para obtener la potencia total.

Por supuesto podemos simplificar este problema utilizando acoplamiento jerárquico formando grupos de 10 AA, y luego acoplando estos grupos en grupos de 10 y así hasta obtener un modelo con 10.000 AAs. Sin embargo esta solución no ayuda de mucho ya que estaremos tratando con un modelo gigante de todos formas (sólo que mejor organizado).

Este tipo de modelos motivan el desarrollo de una extensión vectorial al formalismo DEVS.

Para tratar el caso de que en los modelos vectoriales los distintos componentes pueden tener diferentes parámetros, definiremos primero una extensión a DEVS llamada *Parametrized DEVS*.

4.3.2. Definición de Modelos DEVS Vectoriales

Definimos aquí primero un modelo DEVS parametrizado. Dado un modelo DEVS atómico M obtenemos un *Modelo DEVS Parametrizado*:

$$M(p) = \{X, Y, S, {\delta_{\text{int}}}^p, {\delta_{\text{ext}}}^p, \lambda^p, \text{ta}^p, \}$$

donde $p \in P$ es un parámetro que pertenece a un *conjunto de parámetros* arbitrario tal que $\delta_{\text{int}}, \delta_{\text{ext}}, \lambda$ y ta dependen también de p.

Notar que dos modelos DEVS $M(p_1)$, $M(p_2)$ con $p_1 \neq p_2$ pueden exhibir distintos comportamientos aunque compartan los mismos conjuntos de entrada, salida y de estados (X, Y, y S, respectivamente).

En nuestro ejemplo motivador de los AAs, podemos definir el conjunto de parámetros P como:

$$P = \Re^+ \times \Re^+ \times \Re^+$$

De este modo cada parámetro $p \in P$ tiene la forma (R, C, P_w), estos son, los valores de la resistencia, capacitancia térmica de la habitación y potencia del AA correspondiente.

Con el uso de DEVS Parametrizados el modelador puede definir múltiples modelos DEVS que comparten la misma dinámica pero difieren en algún detalle. Estos modelos serán útiles luego para definir Vectorial DEVS.

4.3.3. Definición de Vectorial DEVS

Dado el modelo escalar DEVS Parametrizado:

$$M(p) = \{X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, \text{ta}, p\}$$

definimos un modelo Vectorial DEVS como la estructura:

$$VD = \{N, XV, YV, P, \{M_i\}\},$$

donde:

- $N \in \mathbb{N}$ es la dimensión del modelo vectorial.
- $XV = X \times Index$ es el conjunto de eventos de entradas vectorial donde X es el conjunto de eventos de entrada del modelo escalar e $Index = \{1, \cdots, N\}$ es el conjunto de índices que indican cuál de los modelos DEVS atómicos recibirá el evento.
- $YV = Y \times Index$ es el conjunto de eventos de salida vectorial donde Y es el conjunto de eventos de salida del modelo escalar e $Index = \{1, \cdots, N\}$ es el conjunto de índices que indica qué modelo escalar de los N, emitió el evento.
- P es un conjunto de parámetros arbitrario.
- Para cada índice $i \in Index$, $p(i) \in P$ es un parámetro y $M_i = M(p(i))$ es el modelo DEVS Parametrizado escalar.

4.3.4. Comportamiento de Modelos Vectorial DEVS

Describimos aquí la semántica de los modelos Vectorial DEVS.

Eventos de Entrada

Cuando un modelo Vectorial DEVS V_d recibe un evento de entrada con valor $(x \in X, i \in Index)$, éste ejecuta la transición externa del modelo DEVS escalar M_i con valor x.

En el caso que el índice sea $i = -1$, se ejecuta la transición externa de *todos* los modelos M_i en forma de *broadcast* con valor x.

Si el índice estuviera fuera del rango correspondiente el evento es descartado.

Eventos de Salida

Cuando el modelo escalar M_i produce un evento con el valor $y \in Y$ el modelo Vectorial DEVS V_D emite un evento de salida con el valor (y, i).

Como dijimos antes, si dos (o más) modelos escalares M_i y M_j pertenecientes a V_D quieren emitir un evento al mismo tiempo el modelo Vectorial DEVS envía el evento de M_i si $i < j$ (y el de M_j en caso contrario). Si M_j no se pasiva por algún evento externo, el modelo Vectorial DEVS procederá a enviar el evento de M_j.

Acoplamiento de Modelos DEVS Vectoriales

Los modelos Vectorial DEVS pueden acoplarse entre ellos como cualquier otro modelo DEVS. Claro está que la dimensión de entrada y salida de ambos bloques deben ser consistentes (sino algunos eventos se descartarán).

Cuando varios modelos Vectorial DEVS $(V_{D_1}, V_{D_2}, \ldots, V_{D_r})$ de dimensión N son acoplados, la estructura obtenida es equivalente a acoplar N modelos DEVS por separado $(M1_1, M2_1, \ldots, Mr_1)$,$(M1_2, M2_2, \ldots, Mr_2)$, etc. Aquí, cada Mk_i es el modelo escalar DEVS con índice i correspondiente al modelo VECDEVS V_{D_k}. Cada sub modelo será idéntico a los otros excepto en sus parámetros iniciales.

Como dijimos antes los modelos Vectorial DEVS pueden ser acoplados como cualquier modelo DEVS. Por ejemplo varios modelos Vectorial DEVS de tamaño N conectado representan N sistemas desconectados, ya que un evento generado en el modelo escalar M_i sólo será propagado a otro modelo escalar M'_i con el **mismo índice**.

Para interconectar componentes de distintos índices debemos introducir nuevos modelos DEVS para enrutar los eventos vectoriales. Todos estos bloques pueden ser pensados como filtros o modificadores que pueden ser incluidos en las conexiones entre modelos Vectorial DEVS.

Volviendo a nuestro ejemplo motivador, si queremos realizar el modelo completo con 10.000 AA, podemos definir un modelo VECDEVS de tamaño 10.000 para cada bloque en la Figura 4.2 y luego conectarlos siguiendo el mismo diagrama de bloques de la figura.

La Figura 4.3 muestra una representación gráfica del modelo Vectorial DEVS en PowerDEVS. Notar que el diagrama y su representación gráfica es similar a la de la Figura 4.2. Aquí cada bloque verde es en realidad un modelo Vectorial DEVS de tamaño 10.000, esto es, cada uno contiene 10.000 instancias del mismo bloque escalar subyacente. Vemos también que hay dos bloques escalares (el bloque fuente para la temperatura de referencia y la temperatura ambiente) conectados a modelos Vectorial DEVS a través de bloques especiales llamados *scalar to vector*.

Introducimos estos bloques debajo.

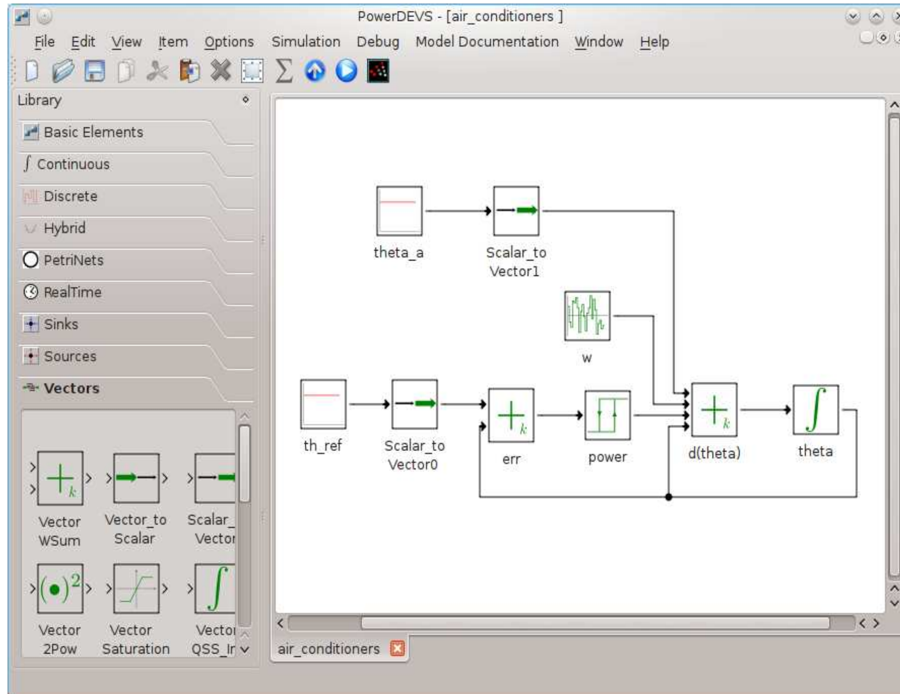

Figura 4.3: PowerDEVS Modelo de 10.000 AA utilizando Vectorial DEVS

4.3.5. Modelos de Interface para Vectorial DEVS

Los modelos Vectorial DEVS puros están limitados a replicar un número arbitrario de instancias de un modelo atómico o acoplado DEVS.

Para representar modelos grandes con estructura compleja necesitamos herramientas que nos permitan interconectar modelos escalares con diferentes índices, y eventualmente, interconectar modelos Vectorial DEVS con modelos DEVS escalares.

Para ello introducimos ahora algunos bloques de interface que cumplen este objetivo. Estos bloques no son modelos Vectorial DEVS, sino simples modelos DEVS escalares que pueden enviar y/o recibir eventos vectoriales.

Bloques de Enrutamiento de Eventos

Si queremos conectar dos Vectorial DEVS para que los eventos emitidos con índice i en el primero sean propagado al segundo con un índice $j \neq i$, necesitamos utilizar un modelo de interfaz DEVS que modifique el índice del evento vectorial. Introducimos tres modelos DEVS con esta funcionalidad.

Index Shift El modelo más simple para manipular modelos DEVS vectoriales es el bloque *Index Shift*. Cuando recibe un evento vectorial con valor (x, i) este bloque envía inmediatamente un evento con el valor $(x, i + sh)$ donde sh es un parámetro entero del bloque.

El bloque index shift tiene de hecho dos puertos de salida. Normalmente

emite los eventos por el primer puerto. Sin embargo si el resultado de sumar $i + sh$ es menor a 1 o mayor que N el evento es emitido por el segundo puerto de salida utilizado para descartar estos eventos. En este caso el índice se modifica de la siguiente forma:

- Si $i + sh < 1$ se emite un evento con índice $(i + sh)\,\%N + N$.
- Si $i + sh > N$ se emite un evento con índice $(i + sh)\,\%N$.

El uso de este bloque se muestra en el modelo de la línea de transmisión descripto en la Sección 4.6.1.

Index Map Un mecanismo más general para enrutar eventos vectoriales puede ser definido con el bloque *Index Map*. El bloque posee como parámetro una matriz de incidencia de $M_{N\times N}$ en la cual $M_{i,j} = 1$ indica que todos los eventos generados por el modelo escalar M_i deben ser propagados al M_j (cambiando el índice de i a j) .

Como esta traducción es una matriz completa el usuario puede definir todas las posibles estructuras de conexión, uno-a-uno, uno-a-muchos, muchos-a-uno, muchos-a-muchos, todos-a-todos, etc.

El uso de este bloque se muestra en el modelo de una red de neuronas pulsantes en la Sección 4.6.3.

Index Selector El modelo *Index Selector* actúa como un filtro de eventos, dejando pasar sólo aquellos eventos que satisfacen un predicado sobre el índice del evento de entrada. Este bloque no cambia ni el valor ni el índice de los eventos sino que descarta los que no cumplen el predicado y re–envía los que sí lo cumplen.

Por ejemplo dado el predicado:

$$Pred((x, i)) = i{>}5 \,\wedge\, i{<}200$$

el bloque *Index Selector* sólo propagará los eventos cuyo índice esté entre 5 y 200. Notar que si el predicado es estático (esto depende de la implementación) es decir, no cambia dinámicamente, el bloque *Index Selector* es una versión especializada del bloque *Index Map* donde la matriz es diagonal y contiene 1s en la fila i, si i satisface el predicado y 0 en caso contrario.

El uso de este bloque también es ilustrado en el modelo de la red pulsante de la Sección 4.6.3.

4.3.6. Conectando Vectorial DEVS con Modelos DEVS Clásicos

Los modelos normales DEVS (esto es, los modelos escalares) no pueden ser conectados directamente con los modelos Vectorial DEVS ya que los conjuntos de entrada y salida de cada uno no son compatibles.

Si queremos conectar un modelo escalar DEVS con un componente escalar de un modelo Vectorial DEVS, necesitamos agregar o eliminar información sobre el índice del valor del evento. Para este fin definimos dos bloques DEVS de interfaz para convertir eventos escalares a vectoriales y vice-versa.

Interfaz Escalar a Vector El bloque *ScalarToVector* es utilizado para convertir eventos escalares en vectoriales. Cuando recibe un evento escalar con valor x envía un evento vectorial con valor (x, i) donde i es un parámetro del bloque que indica a qué índice se debe dirigir el evento vectorial.

El parámetro i puede ser -1 lo que especifica que el evento escalar x debe ser enviado como evento vectorial a todos los índices.

Este bloque es utilizado por ejemplo en el modelo de la Figura 4.3 para propagar la temperatura de referencia a todos los componentes escalares de cada AA (su parámetro $i = -1$).

Vector a Escalar El bloque *VectorToScalar* es análogo al *ScalarToVector* y se utiliza para extraer un valor escalar a partir de un evento vectorial. Cuando recibe un evento con valor (x, i) envía el evento escalar x *sólo si* $i = in$, en caso contrario descarta el evento. El entero in es un parámetro del bloque.

El uso de este bloque se muestra en el ejemplo de la línea de transmisión de la Sección 4.6.1 donde es utilizado para graficar la trayectoria producida por un componente escalar de un integrador vectorial.

4.4. Implementación PowerDEVS de Modelos Vectorial DEVS

Desarrollamos una biblioteca de modelos Vectorial DEVS junto con los modelos de interfaces dentro de la herramienta de simulación PowerDEVS. Describimos aquí esta biblioteca.

La herramienta PowerDEVS implementa tanto modelos atómicos como modelos acoplados DEVS. Los modelos atómicos son definidos como clases de C++ que son instanciadas en tiempo de ejecución (ver Sección 3.1).

Los modelos acoplados no son traducidos a clases C++ sino a una función que instancia cada modelo hijo y los conecta (programáticamente) entre ellos.

En PowerDEVS los eventos de entrada y salida son instancias de una clase C++ llamada `Event` en la cual el valor es de tipo `void *value` (puntero sin tipo), permitiendo al usuario implementar su propio protocolo para el valor de los eventos.

Todos los modelos escalares de la biblioteca transmiten señales continuas discretizadas por QSS. Éstos utilizan la siguiente convención para los eventos de entrada y salida: El puntero `value` apunta a un arreglo `double y[10]` donde `y[0]` es el valor de la señal en el instante actual, `y[1]` representa su derivada, `y[2]` su segunda derivada, etc.

4.4.1. Implementación de los Eventos

Para implementar los eventos vectoriales dejaremos el arreglo de flotantes utilizado por la biblioteca de QSS y agregaremos un índice entero de modo que el valor de un evento vectorial apuntará a un objeto de la clase:

```
class vector {
  double value[10];
  int index;
}
```

Como los arreglos de C++ están indexados comenzando desde 0, los valores de `index` están dentro del rango 0 a $N-1$ en vez de 1 a N como en la definición formal.

Los eventos de PowerDEVS (tanto escalares como vectoriales) poseen también un parámetro *puerto* que representa *por cuál puerto DEVS* ha llegado un evento (en caso de una transición externa) o por cuál puerto DEVS debe ser enviado el evento (en la función λ). Estos puertos también están numerados comenzando desde 0 hasta la cantidad de puertos de entrada/salida.

No deben confundirse el índice de un evento vectorial con el puerto DEVS de un evento. Por ejemplo en la Figura 4.5 cuando el bloque *Vector QSS_Int 1* emita un evento vectorial por su puerto de salida 0 (el único que posee) con índice i, este evento será propagado al primer puerto de entrada 0 del bloque *Vector QSS_Int 0* con el mismo índice i y también al segundo puerto de entrada (1) con índice $i+sh$ (donde sh es el parámetro del *Index Shift (1)*)

4.4.2. Implementación de los Modelos Vectorial DEVS

Implementamos el formalismo Vectorial DEVS en PowerDEVS como un modelo atómico especial que tiene muchos atómicos como estado.

Formalmente, dado un conjunto de modelos parametrizados:

$$M_i = M(p_i) = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, \text{ta}, p_i)$$

donde $p_i \in P$ es un parámetro donde $i \in Index = \{1, \ldots, N\}$,

el modelo vectorial de dimensión N asociado tiene la siguiente descripción:

$$V_M = (V_X, V_Y, V_S, V_{\delta_{\text{int}}}, V_{\delta_{\text{ext}}}, V_\lambda, V_{ta})$$

donde:

- $V_X = X \times Index$
- $V_Y = Y \times Index$
- $V_S = \{(s_1, e_1), (s_2, e_2), \ldots, (s_N, e_N)\}$
- $V\delta_{\text{int}}(s) = \{(s'_1, e'_1), (s'_2, e'_2), \ldots, (s'_N, e'_N)\}$

  donde $(s'_i, e'_i) = \begin{cases} (s_j, e_j + V_{ta}(s)) \text{ si } i^* \neq j \\ (\delta_{\text{int}}(s_j, p_j), 0) \text{ si } i^* = j \end{cases}$
- $V\delta_{\text{ext}}(s, (x, i), e) = \{(s'_1, e'_1), (s'_2, e'_2), \ldots, (s'_N, e'_N)\}$

  donde $(s'_i, e'_i) = \begin{cases} (s_j, e_j + e) \text{ si } i^* \neq j \\ (\delta_{\text{ext}}(s_j, x, e_j + e, p_j), 0) \text{ si } i^* = j \end{cases}$
- $V_\lambda(s) = (\lambda(s_{i^*}, p_{i^*}), i^*)$
- $V_{\text{ta}}(s) = \text{mín}_i(\text{ta}(s_i) - e_i)$

donde $i* = argmin_i(\text{ta}(s_i) - e_i)$, i.e., es el índice del componente escalar que realiza la próxima transición.

Notar que todas las funciones son sólo llamadas a la función escalar correspondiente, por lo cual su implementación es muy simple. Nos queda resolver dos cuestiones:

- Un método para encontrar el mínimo i^* entre todos los modelos escalares.
- Los modelos escalares deben ser parametrizados al inicializarlos.

Como los modelos acoplados de PowerDEVS ya implementan mucha de la funcionalidad deseada para los modelos vectoriales (encontrar el mínimo ta, ejecutar su δ_{int}, δ_{ext}, λ, etc) utilizamos modelos acoplados para representar el estado de los modelos Vectorial DEVS. En este caso los modelos acoplados sólo funciona como una cáscara vacía para contener N modelos atómicos. Este modelo acoplado no posee conexiones de ningún tipo ($EIC = EOC = IC = \{\}$) y sus conjuntos de entrada salida son $X_N = XV, Y_N = YV$ mientras que la función *Select* se define para elegir al mínimo índice entre los atómicos con $ta(s) = 0$.

Este modelo acoplado es creado e inicializado incluyendo los modelos escalares programáticamente en la función *init* del modelo vectorial donde se crean N nuevos modelos escalares y se los inserta en el modelo acoplado.

Como ejemplo, se muestra aquí la implementación en PowerDEVS de un modelo vectorial que tiene N instancias de un bloque escalar *xpower2* que calcula el cuadrado de su entrada.

```
void vector_pow2::init(double t,...) {
  va_list parameters;
  va_start(parameters,t);
  char *fvar= va_arg(parameters,char*);
  N=getScilabVar(fvar ); //obtain the parameter N (dimension) defined in the GUI
  D0 = new Coupling("CoupledPow2"); //instatiate the coupled model
  Connection **EIC1 = new Connection* [0]; //connections are empty
  Connection **EOC1 = new Connection* [0];
  Connection **IC1 = new Connection* [0];
  Simulator **D1 = new Simulator* [N]; //create an array of N atomic models
  for (int i=0;i<N;i++){
        D1[i] = new xpower2("xpower2i"); //instantiate the atomic models
        D1[i]->init(t);                 //initialize the atomic models
  }
  D0->setup(D1,N,IC1,0,EIC1,0,EOC1,0); //build the coupled model
  D0->init(t);                         //initialize the coupled model
}
double vector_pow2::ta(double t) {
  return D0->ta(t); //this returns the time advance of the coupled model
}
void vector_pow2::dint(double t) {
  D0->dint(t); //this performs dint on the imminent child
}
void vector_pow2::dext(Event x, double t) {
  vector vec1=*(vector*)x.value; //cast the event value as a vector
  int index=vec1.index;
  if ((index>-1)&&(index<N)){
        D0->D[index]->dextmessage(x,t); //send the event to the scalar component
        D0->heap.update(index); //update the heap structure
} else if (index==-1) {
  for (int ind=0;ind<N;ind++){
        D0->D[ind]->dextmessage(x,t); //send the event to all scalar components
        D0->heap.update(ind);
 }
};
}
Event vector_pow2::lambda(double t) {
y= D0->D[D0->transitionChild]->lambdamessage(t); //obtain the imminent child's output event
vec=*(vector*)y.value; //convert it to vector format
vec.index=D0->transitionChild; //add the right index
y.value=&vec;
return y;
}
```

La forma de inicializar cada uno de estos N modelos escalares será discutida en la siguiente sección.

4.4.3. Inicialización del los Modelos Vectorial DEVS

Lo único que resta por hacer es definir cómo se inicializa cada modelo DEVS parametrizado dentro un modelo Vectorial DEVS. En PowerDEVS los modelos pueden recibir sus parámetros desde el entorno de Scilab (ver Sección 3.4).

En general cada modelo Vectorial DEVS define su propio conjunto de parámetros pero siguiendo la siguiente convención:

Escalar si el valor del parámetro (evaluado en Scilab) es un valor escalar (por ejemplo 4,5) todos los modelos dentro del Vectorial DEVS recibirán el mismo valor.

Vector Si el valor es un vector V de tamaño N (también evaluado en Scilab) por ejemplo *[1,2,3,4]* el modelo Vectorial DEVS inicializará cada modelo escalar M_i con el valor escalar correspondiente $V(i)$.

4.4.4. Creando un Modelo Vectorial DEVS

Para convertir un modelo escalar de PowerDEVS a vectorial uno debe seguir los siguientes pasos:

1. Definir un nuevo modelo atómico A. Este modelo debe tener el mismo número de puertos de entrada/salida que el modelo escalar.

2. A tendrá un modelo acoplado y el entero N como estado. Este modelo acoplado se crea en la inicialización del atómico A. Allí se crean también N modelos escalares, se los inicializa con sus parámetros como discutimos previamente, y finalmente se los inserta en el modelo acoplado. Este se inicializa *sin conexiones*.

3. La transición interna δ_{int} y la función de avance de tiempo ta son el resultado de evaluar las funciones correspondientes en el acoplado.

4. En la función externa δ_{ext} de A, cuando se recibe un evento con valor (x, i), éste es descompuesto en un evento escalar x y enviado al modelo M_i dentro del acoplado. Si i es igual -1 el evento x es enviado a todos los modelos M_j iterativamente.

5. En la función de salida λ el evento (x, i) es calculado como $x = coupled.lambda()$ e i es el índice del modelo M_i dentro del acoplado que realiza la transición. Como dijimos antes, es el modelo acoplado el que se encarga que la transición interna y función de salida sean las del submodelo M_i con el mínimo tiempo de avance ta.

Este es el procedimiento utilizado por ejemplo para convertir el modelo escalar *xpower2* en su versión vectorial visto en la Sección 4.4.2. Para más ejemplos de un modelo Vectorial DEVS referirse al código fuente de la biblioteca vectorial de PowerDEVS disponible en [77].

4.5. Particionado Automático de modelos Vectorial DEVS

En esta Sección presentamos un algoritmo para el particionado automático de modelos Vectorial DEVS. El objetivo es obtener un sistema compuesto por submodelos que luego puedan simularse concurrentemente en distintas unidades de procesamiento reduciendo así el tiempo de simulación.

Aquí sólo mostramos un algoritmo de particionado mientras que en el Capítulo 5 presentamos técnicas para la simulación en paralelo de estos modelos.

Los modelos DEVS en general ya están divididos en submodelos atómicos y acoplados interconectados, aunque utilizar esta partición para la simulación en paralelo no es conveniente ya que el balance de la carga computacional dependerá de cuánto computa cada modelo. Por lo tanto conviene agrupar varios modelos atómicos en un submodelo de modo que haya más actividad en cada unidad de procesamiento.

En el caso general no es tan simple dividir un modelo agrupando varios submodelos debido a que esto involucra analizar la estructura de las conexiones entre ellos, introducir una cantidad de puertos de entrada y salida y aún así no queda claro qué subsistema físico representa cada una de las particiones.

En el caso de los modelos Vectorial DEVS la división resulta más sencilla ya que un modelo vectorial es en realidad una réplica de N modelos idénticos.

4.5.1. Algoritmo de Particionado

La simulación de DEVS en paralelo (PDES) se basa usualmente en dividir un modelo grande en p submodelos (donde p es el número de procesadores a utilizar) para que cada submodelo tenga una carga computacional similar y la comunicación de eventos entre distintos subsistemas sea minimizada para evitar los costos de la comunicación y de la sincronización. Las técnicas de paralelización DEVS como las que veremos en el Capítulo 5 requieren una etapa previa de particionado la cual, en presencia de estructuras de conexión complejas, pueden resultar muy complicadas. Los modelos puramente Vectorial DEVS consisten en N modelos DEVS escalares idénticos y desconectados. Por lo tanto, en este caso, el particionado es sencillo. Podemos asignar a el primer procesador todos los bloques con índices desde 1 a N/p. Luego, asignamos al segundo procesador todos los bloques con índices desde $N/p + 1$ a $2 * N/p$ y así con el resto de los procesadores. De este modo ningún evento será transmitido entre los procesadores y la simulación en paralelo es fácil de realizar. Sin embargo, en la mayoría de las aplicaciones trataremos con modelos que no son puramente vectoriales. Allí, la presencia de index shift y otros modelos DEVS de interface, producen estructuras más complejas y el método de particionado ya no es tan simple.

En esta Sección, estudiamos el problema de particionado automático de modelos Vectorial DEVS, comenzando con el caso trivial de un modelo puramente vectorial y luego extendiendo el resultado en la presencia de index shift e interfaces. Describimos también la implementación de esta metodología en PowerDEVS.

Particionado de Modelos Puramente Vectorial DEVS

Como mencionamos antes, la idea básica para partir un modelo puramente Vectorial DEVS en p submodelos es asignar los componentes escalares cuyo índice pertenece a cierto rango, al mismo procesador.

Esto es equivalente a convertir un Vectorial DEVS model de dimensión N en p modelos Vectorial DEVS idénticos de dimensión N/p. Por simplicidad, supondremos que p divide a N. [1]

Formalmente, dado un modelo M, que es el resultado de acoplar K modelos Vectorial DEVS $V_1, V_2, \cdots, V_K$ de dimensión N, el algoritmo de particionado consiste en crear p modelos acoplados *desconectados* idénticos a M excepto por que cada uno de ellos tendrá dimensión N/p. Este procedimiento se ilustra para $p = 2$ en la Figura 4.4.

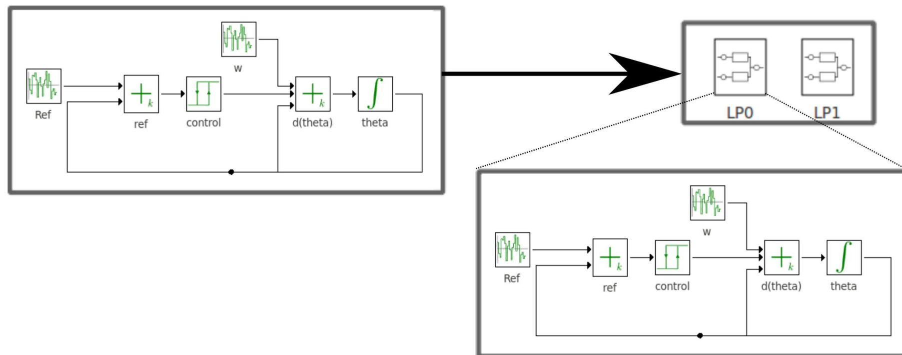

Figura 4.4: Particionado de un modelo acoplado con Vectorial DEVS.

4.5.2. Particionado en Presencia de Modelos DEVS de Interface

Los modelos de *enrutamiento de eventos* provocan conexiones entre componentes escalares con distintos índices. Por ello, ante la presencia de estos bloques, el modelo particionado lucirá como el de la Fig. 4.4 pero con conexiones entre los distintos acoplados (LP0, LP1, etc).

Además, cuando los modelos Vectorial DEVS son interconectados con modelos DEVS clásicos utilizando los modelos de interface (*vector to scalar* y *scalar to vector*), el algoritmo de particionado debe replicar el modelo DEVS escalar en uno o más procesadores.

Describimos ahora las distintas reglas que extienden el algoritmo de particionado para tratar con los bloques de *enrutamiento de eventos* y de interfaces.

Particionado con Bloques Index Shift

El modelo Index Shift es el bloque de enrutamiento de eventos más simple. Interconecta componentes escalares con índice i con aquella de índice $i+sh$ (donde sh es un entero positivo o negativo).

[1] Si N no es múltiplo de p, entonces algunos submodelos tendrán una dimensión mayor a otros para tomar en cuenta el resto de la división.

Sunpodremos que $|sh| > n/P$, i.e., el desplazamiento de índice es menor que el número de componentes escalares en un procesador. En modelos grandes, esta suposición siempre será cierta. Los componentes escalares asignados al procesador j sólo pueden estar conectados con los componentes escalares del procesador $j-1$ o $j+1$.

Luego, para cada bloque *index shift* simplemente agregamos un puerto de entrada y salida a cada acoplado. Cuando $sh > 0$ conectamos el puerto de salida de cada modelo acoplado con el puerto de entrada del siguiente (i.e. conectamos el procesador j con el $j+1$). En el caso contrario conectamos el puerto de salida con el puerto de entrada del anterior (i.e. conectamos el procesador j con el $j-1$).

Internamente, el puerto de entrada recién agregado es conectado a todos los modelos donde estuviera conectado el primer puerto de salida del index shift. Por otro lado el puerto de salida recién agregado es conectado con el *segundo puerto* de salida del index shift en cuestión. De este modo, los eventos con índice mayor a N/p o menor a 1 serán transmitidos a los acoplados anterior o posterior y recibidos allí por el componente escalar correcto.

Por ejemplo, el modelo de la Figura 4.5 contiene un bloque *index shift* ('Index Shift 0') con parámetro $sh = 1$ conectando un integrador vectorial y un sumador vectorial y otro con $sh = -1$ ('Index Shift 1').

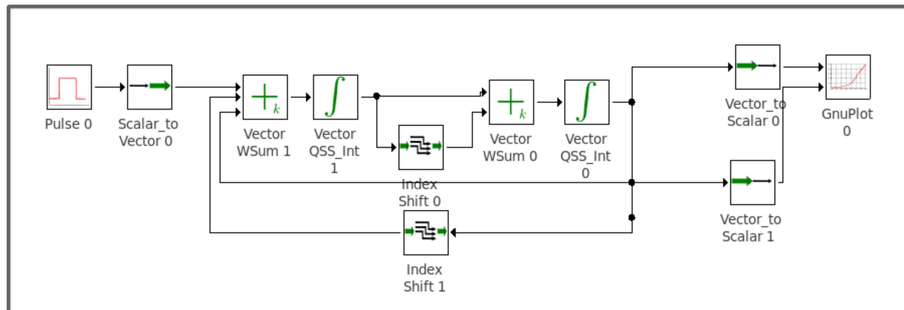

Figura 4.5: Un modelo con bloques de enrutamiento de eventos.

Este modelo puede ser particionado como lo muestra la Figura 4.6. En este nuevo modelo, ambos submodelos son iguales al modelo de la Figura 4.5 excepto que la dimensión es $N/2$. Vemos que el submodelo *LP0* se conecta al siguiente submodelo *LP1* debido al index shift con $sh = 1$ y que el submodelo *LP1* se conecta el anterior (*LP0*) por el index shift con $sh = -1$. Notar que el bloque *Index Shift 1* está rotado horizontalmente, por lo cual sus puertos de salida son los de la izquierda mientras que los de entrada los de la derecha.

No es difícil notar que los modelos de las Figs. 4.6 y 4.5 tienen comportamientos idénticos.

Podemos aplicar reglas similares a los restantes bloques de enrutamiento de eventos (index map e index selector). Sin embargo, esas reglas son más complejas y no serán discutidas aquí.

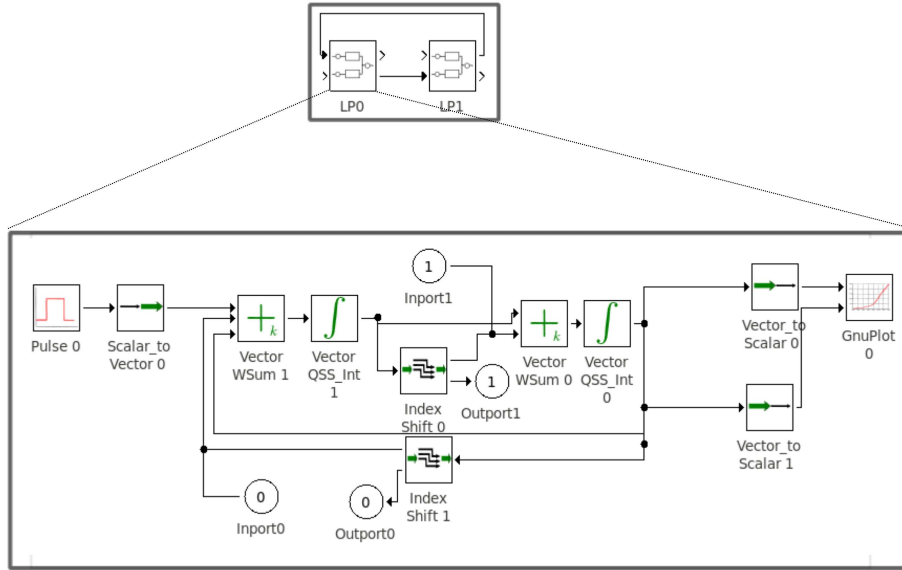

Figura 4.6: Modelo particionado con bloques de enrutamiento de eventos.

Interconexión con Bloques Escalares

La interconexión de modelos Vectorial DEVS y modelos DEVS escalares se realiza a través de los bloques de interfaces *Scalar to Vector* y *Vector to Scalar*

Ante la presencia de estos bloques, el algoritmo de particionado replica todos los bloques escalares e interfaces en todos los submodelos, modificando el parámetro del bloque interfaz para obtener un comportamiento equivalente.

Analicemos primero el caso de la interfaz *Scalar to Vector*, que convierte los eventos escalares en eventos vectoriales con índice i (siendo i el parámetro del bloque). Cuando $i = -1$, el bloque actúa como un *broadcaster* que envía el evento escalar recibido a todos los componentes escalares del modelo vectorial. Por lo tanto, en este caso, no es necesario modificar el parámetro.

Cuando el parámetro es $i \neq -1$ los eventos escalares recibidos sólo son enviados al componente escalar i-ésimo. Luego, modificamos el parámetro del bloque *Scalar to Vector* replicado en el procesador k-ésimo como sigue:

$$i_k = i - (k - 1) \cdot N/p$$

De este modo, en el primer procesador, el parámetro es $i_1 = i$, en el segundo procesador el parámetro es $i_2 = i - N/p$, etc. Por lo tanto, ese parámetro sólo tiene un valor válido cuando $1 \geq i_k \geq N/p$, lo cual sólo se cumple en el procesador donde el componente i-ésimo original fue ubicado.

En los otros procesadores el valor o es negativo o es mayor que N/p por lo cual los eventos son descartados. Así, los eventos escalares son sólo transmitidos al componente escalar correcto.

Para las interfaces *Vector to Scalar* el procedimiento es análogo y sus parámetros son modificados siguiendo las mismas reglas que antes.

4.5.3. Algoritmo de Particionado de Modelos Vectorial DEVS

Resumimos aquí el algoritmo de particionado para un modelo acoplado M con modelos Vectorial DEVS (de tamaño N) en p sub–modelos.

1. Creamos p modelos acoplados $M_1, M_2, \cdots, M_p$, idénticos al acoplado M original.

2. Ajustamos el tamaño de cada modelo vectorial en M_i de N a N/p.

3. Para cada bloque *Index Shift*, agregamos un puerto de salida y uno de entrada a cada acoplado M_i. Luego conectamos el segundo puerto de salida del *Index Shift* al puerto de salida recién agregado. Conectamos también el puerto de entrada a todos los modelos que estuviera conectado el primer puerto de salida del bloque *Index Shift* (ver Fig. 4.6).

   Si el parámetro del bloque *Index Shift* es $sh > 0$, agregamos una conexión del nuevo puerto de salida de M_i al nuevo puerto de entrada M_{i+1} para todo $i < p$. En caso contrario, si $sh < 0$ agregamos una conexión del nuevo puerto de salida de M_i al nuevo puerto de entrada de M_{i-1} para todo $i > 1$.

4. Para cada bloque de interface *ScalarToVector* o *VectorToScalar* dentro de los modelos M_k con parámetro de índice $i \neq -1$, modificamos el índice de acuerdo a:
$$i_k = i - (k-1) \cdot N/p$$

4.6. Ejemplos y Resultados

En esta Sección presentamos algunos casos de uso del modelado con Vectorial DEVS para grandes sistemas.

Primero veremos un modelo de una línea de transmisión LC (circuito resonante compuesto por una bobina y un capacitor) utilizando el Método de Líneas [21] el cual resulta en N segmentos cada uno representado con Vectorial DEVS. Luego analizaremos un ejemplo de un control de potencia de un conjunto de N aires acondicionados similar al ejemplo motivador. Finalmente mostramos dos ejemplos, uno de una Red Neuronal Pulsante [88] y otro de la simulación de un sistema dinámico en Espacio de Estados.

4.6.1. Línea de Transmisión LC

El siguiente sistema de ecuaciones representan un modelo a parámetros concentrados de una línea de transmisión formada por N secciones de circuitos LC:

$$\begin{aligned}\dot{v}_j =& \frac{i_j - i_{j+1}}{C}\\ \dot{i}_j =& \frac{v_{j-1} - v_j}{L}\end{aligned}$$

para $i = 1, \ldots, N$.

Consideramos también un pulso de entrada:

$$v_0(t) = \begin{cases} 1 & \text{si } t < 1 \\ 0 & \text{caso contrario} \end{cases} \quad (4.2)$$

Tomando $N = 600$ y utilizando parámetros $L = C = 1$ y condiciones iniciales nulas, obtenemos un sistema marginalmente estable de orden 1200.

Este sistema es fácilmente representable utilizando Vectorial DEVS. De hecho es el que hemos visto en la Figura 4.5.

Simulamos el sistema con QSS3, la Figura 4.7 muestra el resultado del voltaje al final de la línea de transmisión $v_{600}(t)$.

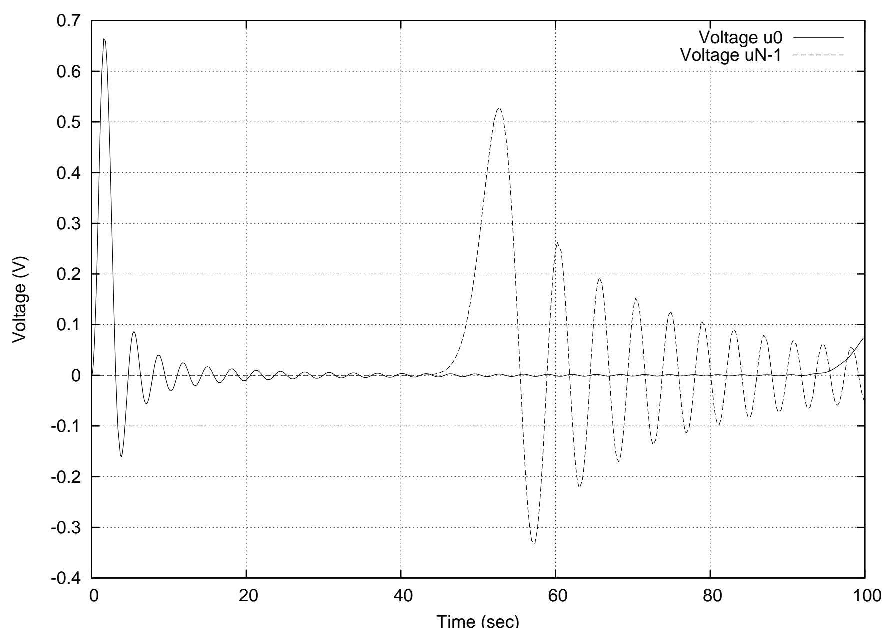

Figura 4.7: Resultado de simulación de la línea de transmisión.

4.6.2. Control de Energía de un Conjunto de Aires Acondicionados

Este ejemplo, tomado de [75] es un modelo para estudiar la dinámica y control del consumo energético de un grupo de equipos de Aire Acondicionados (AA).

Consideramos aquí un gran número de equipos de aire acondicionado utilizados para controlar la temperatura de distintas habitaciones. La temperatura en la habitación i–ésima $\theta_i(t)$ sigue la ecuación:

$$\frac{d\theta_i(t)}{dt} = -\frac{1}{C_i \cdot R_i}[\theta_i(t) - \theta_a + R_i \cdot P_i \cdot m_i(t) + w_i(t)], \quad (4.3)$$

donde R_i y C_i son parámetros que representan la resistencia y capacitancia térmica de la habitación i respectivamente. P_i es la potencia del equipo de aire

acondicionado i cuando éste está encendido. θ_a es la temperatura del ambiente y $w_i(t)$ es un término de ruido que representa la perturbaciones térmicas.

La variable $m_i(t)$ es el estado del i–ésimo equipo de aire acondicionado, que toma como valor 1 cuando el equipo está encendido ó 0 en caso contrario. Sigue una ley prendido-apagado de control con histéresis:

$$m_i(t^+) = \begin{cases} 0 & \text{si } \theta_i(t) \leq \theta_r(t) - 0{,}5 \text{ y } m_i(t) = 1 \\ 1 & \text{si } \theta_i(t) \geq \theta_r(t) + 0{,}5 \text{ y } m_i(t) = 0 \\ m_i(t) & \text{caso contrario} \end{cases} \tag{4.4}$$

donde $\theta_r(t)$ es la temperatura de referencia calculado por un sistema de control global.

El consumo energético del grupo completo de aires acondicionado se calcula como:

$$P(t) = \sum_{i=1}^{N} m_i(t) \cdot P_i$$

y el sistema de control global lo regula para seguir un "perfil" energético $P_r(t)$.

Para lograr esto se utiliza una ley de control Proporcional Integral (PI) que calcula la temperatura de referencia como:

$$\theta_r(t) = K_P \cdot [P_r(t) - P(t)] + K_I \cdot \int_{\tau=0}^{t} [P_r(\tau) - P(\tau)] \mathrm{d}\tau$$

donde K_P y K_I son parámetros del controlador PI.

En este ejemplo utilizamos $N = 2400$ equipos de aire acondicionados y utilizamos los parámetros dados en [75].

Luego, estamos trabajando con un sistema híbrido de gran escala que consiste en 2401 ecuaciones diferenciales con 2400 condiciones de cruce por cero.

En la Figura 4.8 vemos el modelo realizado con Vectorial DEVS mientras que en la Figura 4.9 se muestra el resultado de simulación del sistema.

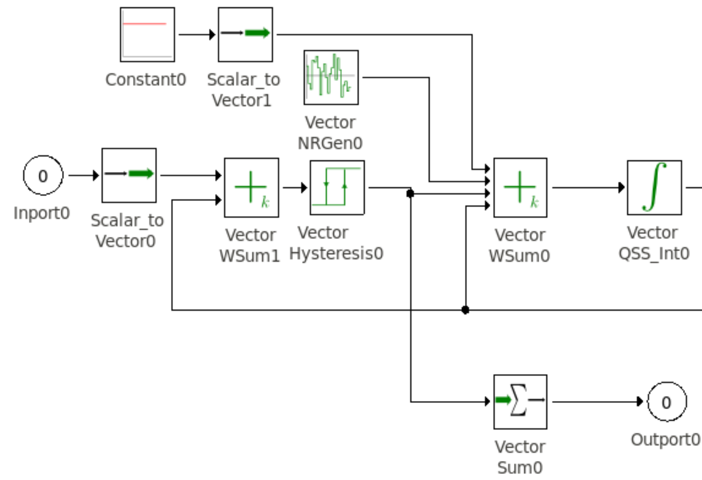

Figura 4.8: Modelo del grupo de AA.

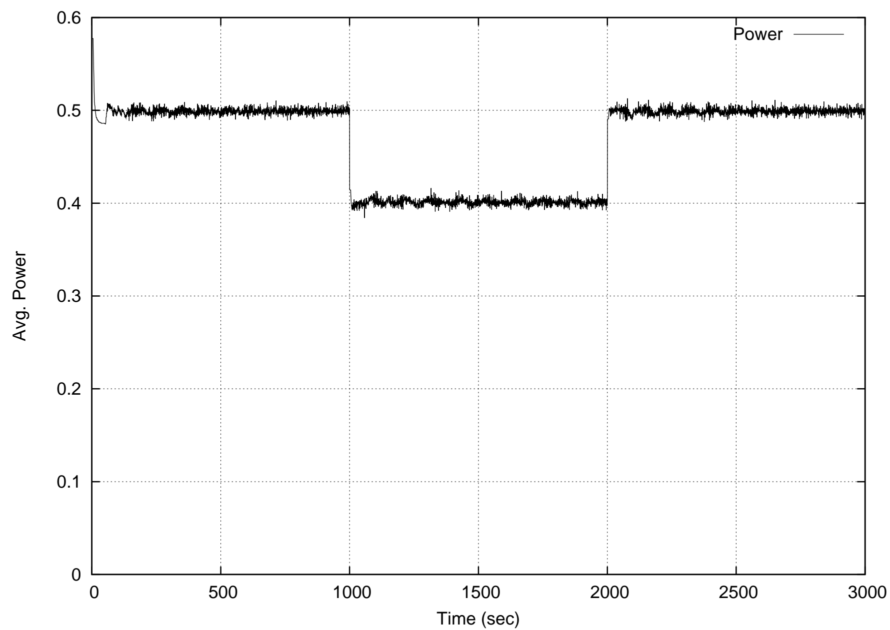

Figura 4.9: Consumo energético del sistema AA.

4.6.3. Redes Neuronales Pulsantes

Este ejemplo tomado de [88] donde se simula una Red Neuronal Pulsante (o SNN por su siglas en inglés) utilizando los métodos de QSS.

El modelo está compuesto de 1000 neuronas del tipo *leaky integrate-and-fire* donde cada neurona sigue la dinámica:

$$\frac{dV}{dt} = (V_{rest} - V) - g_{ex}(E_{ex} - V_{rest}) + g_{inh}(E_{inh} - V_{rest}) \tag{4.5}$$

La neurona se dispara cuando cruza el umbral de $-50mV$ y reinicializa su voltaje a $V_{rest} = -60mv$. g_{ex} y g_{inh} son las conductancias sinápticas excitatorias e inhibitorias las cuales siguen un decaimiento exponencial:

$$\tau_{ex}\frac{dg_{ex}}{dt} = -g_{ex} \tag{4.6}$$

y

$$\tau_{inh}\frac{dg_{inh}}{dt} = -g_{inh} \tag{4.7}$$

Las neuronas están conectadas entre ellas aleatoriamente con una probabilidad de conexión del 2 %.

Modelamos la red utilizando Vectorial DEVS como puede ser visto en la Figura 4.10.

Los bloques *map_exc* y *map_inh* son instancias de *Index Map* (Sección 4.3.5) y representan cómo las neuronas está conectadas.

Los bloques *excitatory* y *inhibitory* son instancias de *Index Selector* y separan la red en neuronas excitatorias e inhibitorias con una relación de 4 : 1 respectivamente.

En la Figura 4.11 vemos una red de 1000 neuronas representadas con Vectorial DEVS (Ecuación 4.5). La Figura 4.12 muestra el modelo Vectorial DEVS de las corrientes sinápticas (Ecuaciones 4.6 y 4.7).

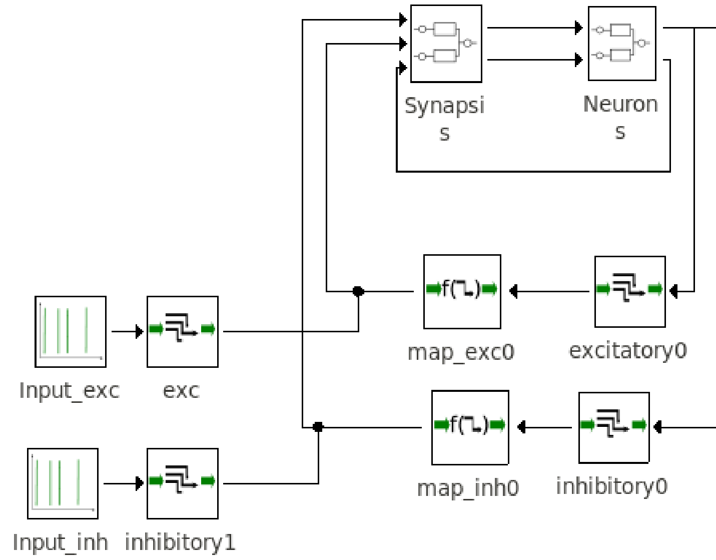

Figura 4.10: Modelo de la Red Neuronal Pulsante.

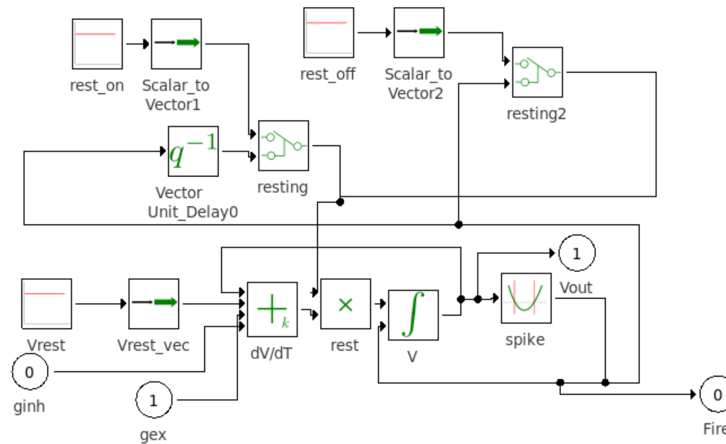

Figura 4.11: Modelo de una neurona.

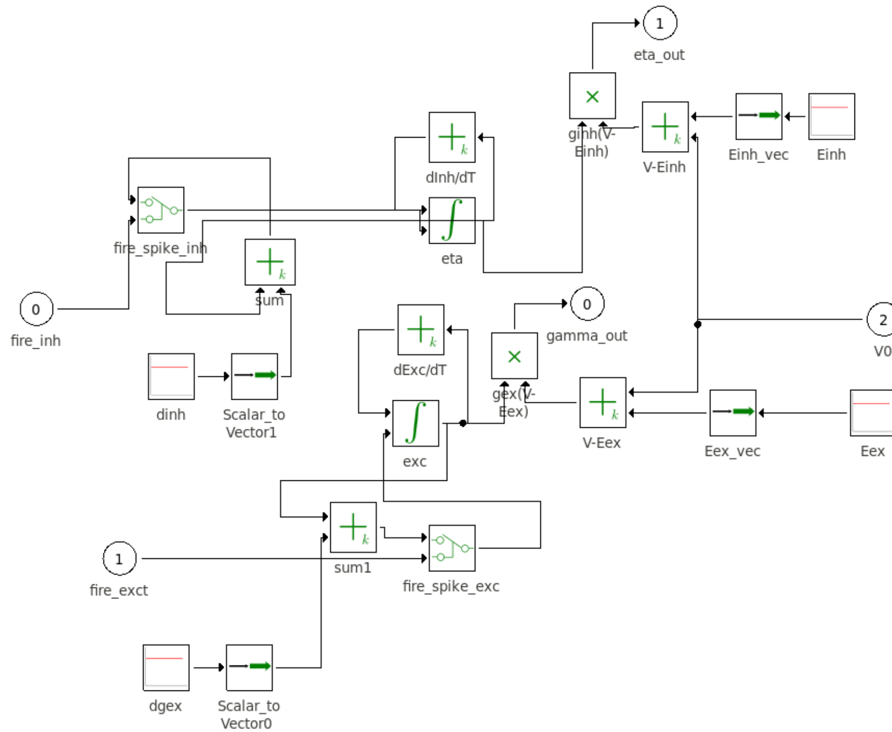

Figura 4.12: Modelo de las sinápsis.

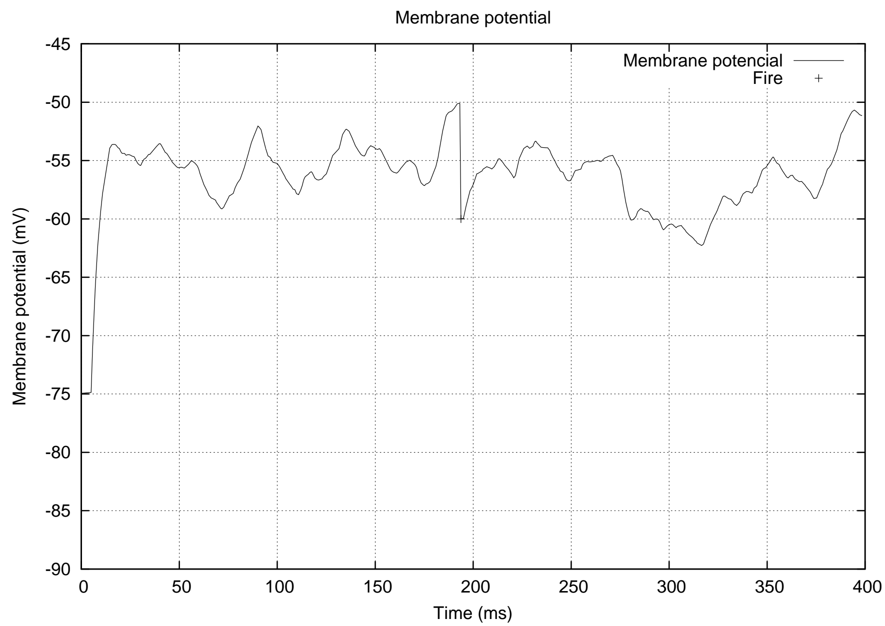

Figura 4.13: Potencial de la membrana de una neurona elegida al azar.

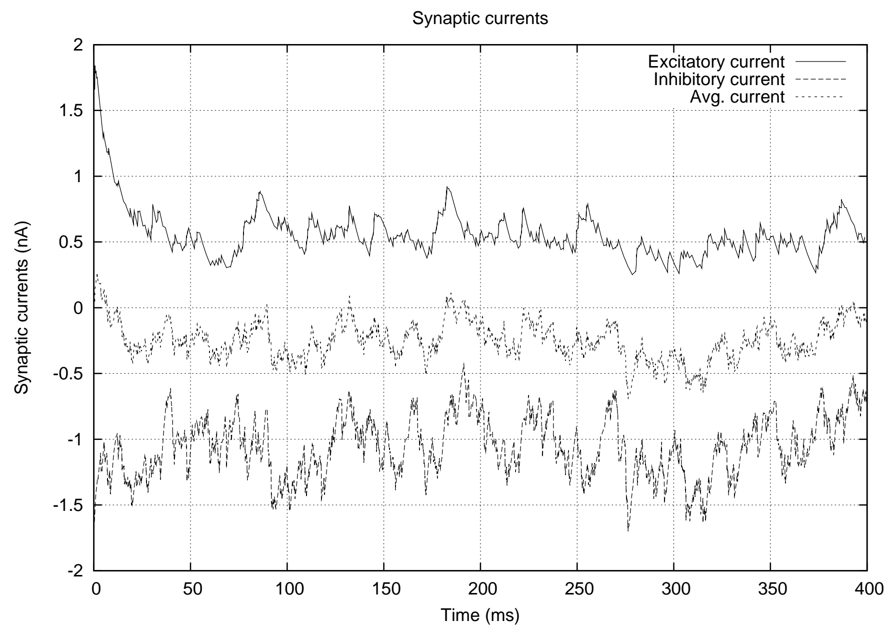

Figura 4.14: Corrientes sinápticas de una neurona elegida al azar.

4.6.4. Modelo en Espacio de Estados

En este ejemplo presentamos un sistema dinámico lineal y estacionario genérico representado en espacios de estado como:

$$\dot{x}(t) = Ax(t) + Bu(t)$$

donde $x(t)$ es el vector de estado, $u(t)$ es el vector de entradas y A y B son matrices.

Aquí utilizamos modelos Vectorial DEVS para los integradores y el sumador y bloques *Matrix Gain* para las matrices A y B. En la Figura 4.15 vemos el modelo de PowerDEVS.

Mientras que gráficamente es compacto, este modelo puede representar cualquier sistema lineal y estacionario de N ecuaciones diferenciales.

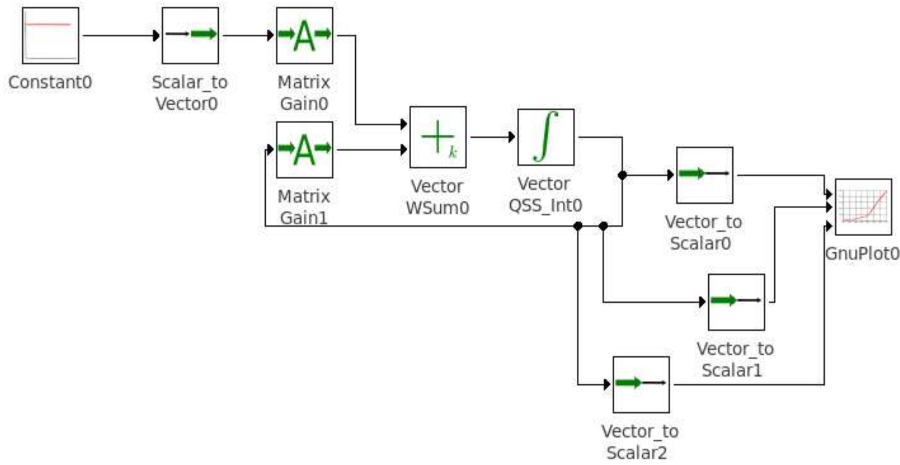

Figura 4.15: Modelo en Espacio de Estados.

La implementación en PowerDEVS de todos estos bloques permite utilizar parámetros genéricos desde Scilab. En este caso definiendo por ejemplo en Scilab las siguientes matrices:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -2 & -4 & -3 \end{bmatrix}, \ B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{4.8}$$

para una entrada constante $u = 1$ se obtienen las trayectorias de la Figura 4.16 para las tres variables de estado.

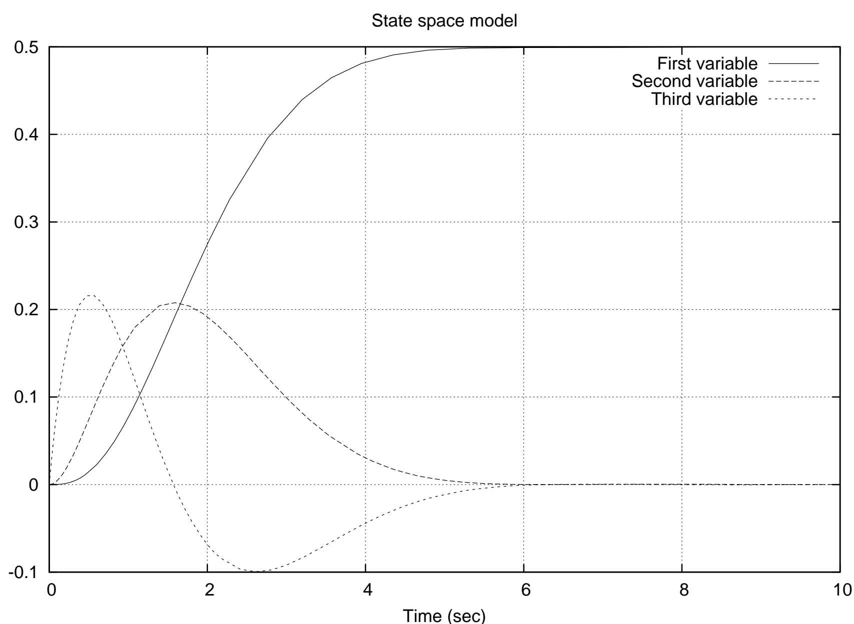

Figura 4.16: Trayectorias de estado.

4.7. Conclusiones

En este Capítulo introdujimos una extensión al formalismo DEVS llamado Vectorial DEVS que apunta al modelado gráfico de grandes sistemas continuos e híbridos. Al extender la notación del formalismo DEVS permitimos la descripción de arreglos de modelos que provee también mecanismos para conectarlos entre sí.

Implementamos Vectorial DEVS en PowerDEVS, incluyendo nuevos bloques (integradores, sumadores, filtros, etc) y desarrollamos varios casos de estudio que muestran Vectorial DEVS en acción.

La estructura regular de los modelos Vectorial DEVS permiten particionarlos fácilmente para luego ser simulados en paralelo. En la Sección 4.5 de este Capítulo vimos un algoritmo para el particionado estático el cual utilizaremos para realizar simulación en paralelo en el próximo Capítulo.

Capítulo 5

Simulación en Paralelo - SRTS y ASRTS

La Simulación de Eventos Discretos en Paralelo (o PDES por sus siglas en inglés) es una técnica que puede ser utilizada para simular grandes modelos DEVS en una arquitectura multi–core o en un cluster. En PDES, el modelo es primero dividido en varios sub–sistemas llamados "procesos físicos". Luego los procesos físicos son simulados concurrentemente en distintos procesadores lógicos (o LP).

En comparación a simular un modelo grande de forma secuencial, PDES reduce el costo computacional pero introduce un nuevo problema relacionado a la necesidad de sincronizar las distintas simulaciones. Se requiere la sincronización entre los distintos Procesadores Lógicos ya que cada sub–simulación necesita conocer el resultado de las otras para simular correctamente su propio sub–sistema. Si los Procesadores Lógicos no están sincronizados correctamente la simulación puede recibir eventos fuera de orden (eventos con timestamp menor al tiempo de simulación actual), lo que puede llevar al resultado incorrecto. Esto se llama *restricción causal* sobre los eventos. Como vimos en la Sección 2.7.2, hay muchos algoritmos para solucionar este problema de distintas maneras.

En este Capítulo presentamos una novedosa técnica para PDES especializada para la aproximación QSS de sistemas continuos e híbridos llamada Scaled Real–Time Synchronization (SRTS). La idea básica es sincronizar la simulación de cada Procesador Lógico con una versión escalada del tiempo físico o wall–clock. Como todos los Procesadores Lógicos están sincronizados contra el tiempo físico, los Procesadores Lógicos están indirectamente sincronizados entre ellos. Como este enfoque está pensado para sistemas continuos e híbridos, los eventos representan trayectorias continuas, por lo tanto, los errores de sincronización provocarán error numérico. Si estos errores están acotados, esto no invalida el resultado final de la simulación por lo cual podemos relajar la restricción causal. Mientras que con SRTS el usuario debe proveer el factor de escalado de tiempo real como un parámetro de simulación, desarrollamos una versión Adaptive–SRTS (ASRTS) del algoritmo donde este factor es ajustado dinámicamente dependiendo de la carga del sistema.

Hay distintos niveles de paralelismo, a nivel de instrucción, a nivel de datos, a nivel de tarea, etc. Estos niveles difieren en varios aspectos, como memoria distribuida o compartida, múltiple o mono–procesadores, tipo de comunicación que utilizan, como paso de mensajes o memoria compartida, etc. Nos centraremos

en el paralelismo a nivel de tarea donde cada Procesador Lógico ejecuta un código distinto con distintos datos de entrada, de acuerdo a la taxonomía de Flynn [31] trabajaremos con un flujo Multiple Instructions and Multiple Data (MIMD).

En este trabajo utilizaremos en una arquitectura multi–core [16] ya que ofrece muchas características deseables para nuestra implementación como memoria compartida, comunicación local y un juego de instrucciones genérico.

Implementamos SRTS y ASRTS en PowerDEVS [11] y desarrollamos varios casos de estudio en los cuales obtenemos una aceleración en el tiempo de simulación de hasta nueve veces.

Este Capítulo presenta las dos técnicas SRTS y ASRTS junto con varios casos de estudio en los cuales se analiza la performance de la implementación y finalmente algunas conclusiones.

5.1. Scaled Real Time Synchronization – SRTS

5.1.1. Idea Básica

Como mencionamos antes, diversas técnicas han sido propuestas en la literatura para PDES, basadas en algoritmos conservadores, optimistas o desincronizados.

Los modelos DEVS resultantes de la aplicación de métodos de QSS a sistemas continuos de gran escala poseen las siguientes particularidades:

1. Cada evento se origina en un integrador cuantificado y es transmitido instantáneamente a otro integrador cuantificado a través de funciones estáticas que dependen de la variable de estado correspondiente.

2. Los eventos recibidos por los integradores cuantificados modifican su evolución futura, pero no provocan eventos de salida instantáneamente.

3. Distintos integradores cuantificados provocan eventos de salida en distintos instantes.

4. Cada evento representa una función polinomial a tramos.

Por un lado, la segunda característica implica que la sincronización es generalmente necesaria y técnicas como NOTIME [79] raramente pueden ser usadas. Por otro lado, si la sincronización es tan estricta con el ordenamiento de todos los eventos, habrá pocos cómputos realizados en paralelo. La razón de este hecho es simple. Cada vez que un integrador provoca un evento, el evento debe ser transmitido a algunas funciones estáticas y luego a través de éstas a algunos integradores cuantificados. Por lo tanto hay un ciclo de dependencias por lo cual técnicas como la mencionada no serán de gran ayuda.

Un algoritmo *conservador* no permitirá que ningún otro Procesador Lógico avance el tiempo de simulación hasta que todos esos cálculos hayan sido completados. Por lo tanto no hay ganancia en paralelizarlo. Un algoritmo *optimista* dejará que el tiempo de simulación de los otros Procesadores Lógicos avance, pero en cuanto el efecto del evento sea propagado a ellos, se deberá ejecutar un mecanismo de roll–back. Este mecanismo aplicado a grandes sistemas híbridos es impracticable debido a que consume mucha memoria y tiempo, y además, es difícil estimar a priori, cuánto tiempo utilizará el roll–back.

Para superar estas dificultades, proponemos una técnica en la cual realizamos una sincronización no estricta. Teniendo en cuenta que los eventos representan secciones de un polinomio, los errores de sincronización implican un error numérico acotado en las trayectorias transmitidas de un procesador a otro.

Como en otras técnicas de PDES, la técnica Scaled Real Time Synchronization (SRTS) divide el modelo en sub–modelos, cada uno representando procesos físicos, y cada uno de éstos es simulado en un Procesador Lógico distinto. Para evitar el costo de la sincronización entre procesos, en vez de sincronizar el tiempo de simulación entre todos los Procesadores Lógicos, en SRTS sincronizamos el tiempo de simulación de cada Procesador Lógico con el tiempo físico (o wall–clock). La única comunicación entre distintos Procesadores Lógicos ocurre cuando los eventos de uno son transmitidos al otro. Estos eventos son transmitidos utilizando el mecanismo de IPC *mailbox* de RTAI [26].

Cada Procesador Lógico se sincroniza con el tiempo físico de la siguiente manera: si el próximo evento en el Procesador Lógico está agendado para dentro de τ unidades de tiempo de simulación, el Procesador Lógico *espera* hasta que el tiempo físico avance τ/r unidades de tiempo físico, donde r es el *factor de escalado del tiempo real*. Es un parámetro que debe ser elegido de acuerdo con la velocidad a la cual el sistema puede ser simulado. r determina qué relación mantiene el tiempo de simulación con el tiempo físico.

Una vez que el período de espera termina, el Procesador Lógico calcula su próximo evento de salida y transiciona a un nuevo estado. Cuando este evento de salida necesita ser propagado a sub–modelos pertenecientes a otros Procesadores Lógicos, se envía un mensaje con *timestamp* que contiene el evento al mailbox correspondiente. Durante el período de espera, los Procesadores Lógicos pueden recibir mensajes en su mailboxes. Cuando recibe un mensaje, el Procesador Lógico procesa el evento haciendo uso del timestamp, calcula su próximo estado, y si el próximo evento a emitir está agendado después del tiempo actual, vuelve a esperar.

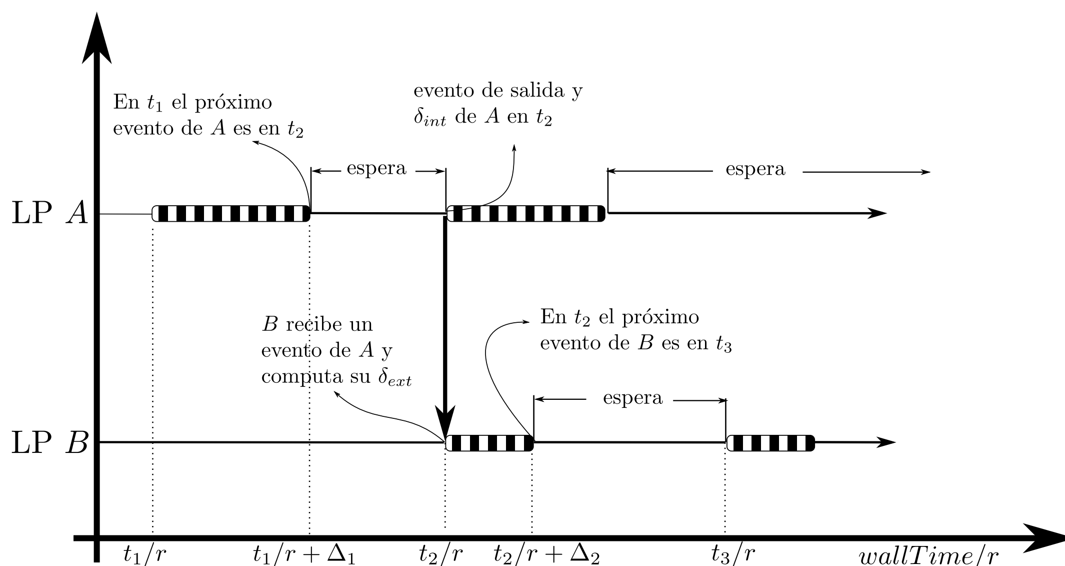

Figura 5.1: Diagrama temporal de la técnica SRTS.

En la Figura 5.1, mostramos un diagrama temporal donde dos Procesadores Lógicos (Procesador Lógico A y Procesador Lógico B) son sincronizados utilizando SRTS. Inicialmente ambos Procesadores Lógicos están esperando. En el tiempo de simulación lógico t_1 (tiempo físico escalado t_1/r), Procesador Lógico A calcula su próximo evento. Este cálculo consume Δ_1 unidades de tiempo físico. Luego el Procesador Lógico A *espera* hasta que t_2 sea alcanzado (a los t_2/r del tiempo físico escalado) momento en el cual emite su evento de salida, hace una transición interna y vuelve a esperar. Este evento es propagado al Procesador Lógico B que calcula su transición externa procesando el evento recibido (consumiendo Δ_2 de tiempo físico), agenda su próximo evento para el tiempo t_3 y vuelve a esperar. Este ciclo es repetido hasta que el tiempo final de simulación es alcanzado.

Utilizando este esquema en un *simulador abstracto*, no puede haber eventos *straggler*[1]; el único problema podría surgir cuando ocurren eventos simultáneos en distintos Procesadores Lógicos, pero esto no tiene consecuencias graves cuando se utiliza QSS ya que no importa el orden en que los eventos simultáneos son procesados. En una implementación real, en cambio, podrán aparecer eventos straggler, ya que ni la sincronización ni la comunicación están libres de latencias, y también el cómputo de cada evento consume tiempo.

De todas formas, como mencionamos antes, un error de sincronización acotado en el contexto de QSS sólo provoca un error numérico acotado en las trayectorias transmitidas entre los Procesadores Lógicos.

5.1.2. Algoritmo Scaled Real Time Synchronization

Como en otras técnicas de PDES, SRTS requiere que el modelo sea primero particionado en tantos procesos físicos como Procesadores Lógicos hay disponibles. En esta Tesis no discutiremos ninguna metodología para realizar este particionado eficientemente más allá de lo que presentamos en la Sección 4.5. Existen muchas publicaciones acerca de este tema en la literatura [18, 25, 39, 83].

Suponemos que cada Procesador Lógico mantiene un sub–modelo DEVS acoplado, y que todos estos están conectados a través de puertos de entrada y salida que modelan la interacción entre los sub–sistemas. La Figura 5.2 muestra la estructura de acoplamiento de un modelo M dividido en cuatro sub–modelos y simulado en cuatro Procesadores Lógicos distintos.

Inicialización de la Simulación

Una simulación secuencial DEVS invoca primero a una rutina de inicialización que asigna los valores iniciales del modelo, sus parámetros, etc. Luego la simulación puede comenzar.

En SRTS, luego de que cada sub–modelo ha sido inicializado, todos los hilos de simulación deben comenzar en el mismo tiempo físico para asegurar la sincronización entre los distintos Procesadores Lógicos.

Para lograr el inicio simultáneo, se utiliza una barrera. Todos los hilos de simulación son lanzados y el hilo i–ésimo comienza como sigue:

1. Asigna la variable compartida $flag[i] = true$.

[1] Un evento straggler es un evento que tiene un timestamp menor al del tiempo de simulación lógico del receptor.

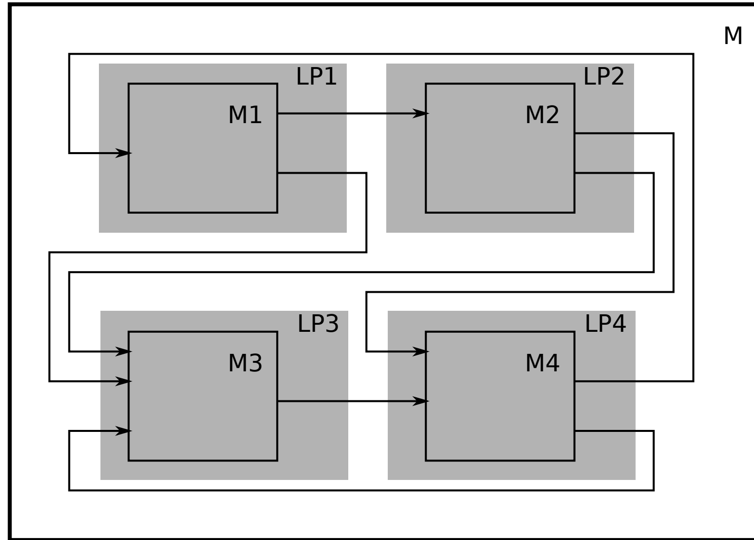

Figura 5.2: Estructura de acoplamiento de SRTS.

2. Espera a que $flag[j] = true$ para todo $j \neq i$.
3. Mide el tiempo físico inicial ($t_0[i]$) del Procesador Lógico.
4. Comienza la rutina de simulación.

El tiempo físico inicial es luego utilizado para calcular el tiempo físico relativo al inicio de la simulación, restándolo al tiempo físico actual.

Algoritmo de Simulación

Describimos ahora, paso a paso el algoritmo que ejecuta cada Procesador Lógico. Aquí se supone que queremos simular el modelo hasta un tiempo final tf. El algoritmo es como sigue:

1. Calcular el tiempo del próximo evento del sub–modelo. A este tiempo lo denominamos t_n. .
2. Si $t_n > t_f$, entonces $t_n := t_f$.
3. Esperar hasta que el tiempo físico escalado alcance el tiempo de simulación (i.e., $t_p/r = t_n$) o hasta que llegue un mensaje.
4. Si se recibe un mensaje, ir al paso 10.
5. Si $t_n = t_f$, la simulación terminó.
6. Avanzar el tiempo de simulación hasta $t := t_n$.
7. Calcular y propagar el evento de salida en el sub–modelo (transición externa).
8. Si el evento debe ser propagado a otros sub–modelos, enviar el mensaje a los Procesadores Lógicos correspondientes.

9. Recalcular el estado del modelo que causó el evento (transición interna) y volver al paso 1.

10. Avanzar el tiempo de simulación t, a el valor del timestamp del mensaje.

11. Propagar el evento dentro el sub–modelo (ejecutando la transición externa) y volver al paso 1.

Este algoritmo funciona bien si los Procesadores Lógicos mandan y reciben los mensajes en el tiempo correcto. Para enviar un mensaje en el tiempo correcto, un Procesador Lógico debe terminar los cálculos correspondientes antes de ese tiempo, i.e., no debería estar en *overrun*.

Política de Overrun

Las situaciones de overrun pueden ser minimizadas utilizando un valor de escalado del tiempo real r suficientemente chico. Por otro lado, queremos que r sea lo más grande posible para simular el sistema más rápido. Incluso eligiendo un r pequeño, los sistemas operativos de tiempo real tienen latencias y las situaciones de overrun ocurren de todas formas.

Cuando un Procesador Lógico está en overrun, continua la simulación pero sin recibir eventos de entrada. La razón es que estos eventos tendrán un timestamp en el futuro con respecto al tiempo de simulación lógico del Procesador Lógico en overrun. Luego, es mejor procesar los eventos de entrada una vez que la situación de overrun haya terminado.

Manejo de Mensajes

Como mencionamos antes, los eventos enviados a sub–modelos pertenecientes a distintos Procesadores Lógicos son transmitidos a través de mensajes con timestamp. Puede ocurrir que un sub–modelo esté en overrun u ocupado calculando las transiciones de estado mientras un mensaje es enviado a él. Si este Procesador Lógico no completa su trabajo y otro Procesador Lógico le envía un nuevo mensaje, tenemos dos opciones. Encolar los mensajes hasta que el Procesador Lógico esté listo para procesarlos, o podemos descartar algunos de los mensajes.

Mientras que en el contexto general de DEVS, encolar los mensajes resulta lo más razonable, no lo es en SRTS donde el objetivo es simular sistemas continuos por lo cual los eventos representan cambios en las trayectorias.

Si en el tiempo t un Procesador Lógico termina su trabajo y ve que hay dos mensajes de entrada en su mailbox esperando a ser procesados, uno con timestamp t_1 y otro con timestamp t_2 donde $t_1 < t_2 < t$, no tiene sentido procesar el mensaje más viejo. Ya es tarde para ello y el cambio que representa ese mensaje ha sido ignorado. Por el contrario sí tiene sentido procesar el cambio en t_2 que contiene información más reciente acerca de la trayectoria.

Por esta razón, cada sub–modelo en SRTS usa un mailbox para cada puerto de entrada (i.e., un mailbox por trayectoria de entrada), donde cada mailbox tiene capacidad unitaria. Sin importar si el Procesador Lógico puede leer o no el mensaje, cada mensaje en el mailbox es sobre–escrito por la llegada de un mensaje nuevo.

5.1.3. SRTS y Error Numérico en los Métodos de QSS

El uso de SRTS implica que los mensajes enviados entre Procesadores Lógicos pueden ser recibidos en tiempos incorrectos. Como los Procesadores Lógicos no pueden recibir mensajes cuando están en overrun, nunca recibirán mensajes con timestamp mayores al tiempo de simulación lógico. En otras palabras, nunca recibirán mensajes del *futuro*. Los errores de sincronización siempre implican que un mensaje llega tarde.

Supongamos que utilizamos SRTS con dos procesadores para simular mediante QSS un sistema de alto orden:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{v}(t))$$

Partimos el sistema como sigue:

$$\begin{aligned}\dot{\mathbf{x}}_a(t) &= \mathbf{f}_a(\mathbf{q_a}(t), \mathbf{q_b}(t), \mathbf{v}(t))\\ \dot{\mathbf{x}}_b(t) &= \mathbf{f}_b(\mathbf{q_a}(t), \mathbf{q_b}(t), \mathbf{v}(t))\end{aligned}$$

donde:

$$\mathbf{x} = [\mathbf{x}_a\ \mathbf{x}_b]^T;\ \mathbf{q} = [\mathbf{q}_a\ \mathbf{q}_b]^T;\ \mathbf{f} = [\mathbf{f}_a\ \mathbf{f}_b]^T$$

El retardo introducido por SRTS implica que $\mathbf{q}_a$ es recibido con un retardo por el Procesador Lógico que calcula $\mathbf{x}_b$, y vice versa. Por lo tanto, la aproximación por QSS toma la forma:

$$\begin{aligned}\dot{\mathbf{x}}_a(t) &= \mathbf{f}_a(\mathbf{q_a}(t), \mathbf{q_b}(t-\tau_b(t)), \mathbf{v}(t))\\ \dot{\mathbf{x}}_b(t) &= \mathbf{f}_b(\mathbf{q_a}(t-\tau_a(t)), \mathbf{q_b}(t), \mathbf{v}(t))\end{aligned}$$

donde $\tau_a(t)$ y $\tau_b(t)$ son los retardos del canal de comunicación entre los LPS (incluyendo latencias, efectos del overrun, y los retardos afectados por el factor de escalado de tiempo real).

Discutiremos más adelante algunos detalles de implementación que aseguran que los retardos están acotados. Dependiendo de algunas características de la ODE original (propiedad de estabilidad entrada–a–estado [20]), la presencia de un retardo acotado sólo provocará error numérico que se superpondrá al error introducido por la aproximación de QSS. Aunque este análisis puede ser fácilmente extendido a sistemas con N Procesadores Lógicos, no pretendemos aquí hacer un estudio formal de estabilidad y de cotas de error numérico relacionado al uso de SRTS. Cabe mencionar sin embargo que el error introducido por SRTS es minimizado cuando los retardos y el overrun son minimizados.

5.1.4. Implementación en PowerDEVS– RTAI

Implementamos el algoritmo de SRTS en PowerDEVS en una arquitectura multicore. Elegimos PowerDEVS porque implementa toda la familia de métodos de QSS y puede ser ejecutado en tiempo real con lo desarrollado en el Capítulo 3. El uso de un RTOS es necesario para SRTS debido a que como mencionamos antes, los retardos introducidos por la latencia deben ser minimizados.

La versión de PowerDEVS para PDES genera un ejecutable que realiza los siguientes pasos:

1. Inicializa el modelo de PowerDEVS (ver Sección 2.5).
2. Inicializa también el sistema de tiempo real de RTAI, configurando relojes de tiempo real y una tarea de tiempo real principal que realiza los siguientes pasos.
3. Para cada LP, crea e inicializa tantos mailboxes como puertos de entrada tenga.
4. Lanza un hilo de tiempo real por cada procesador físico con un parámetro *index* que permite a cada hilo saber cuál es su sub–sistema correspondiente. Estos hilos ejecutarán el bucle de simulación.
5. Espera a que todos los hilos de tiempo real terminen y luego termina después de liberar correctamente todos los mailboxes e hilos creados.

Hemos desarrollado también un live–cd desde el cual un usuario puede probar (y luego instalar) PowerDEVS–RTAI en cualquier procesador de la familia i386.

Sincronización y Asignación de CPU

Implementamos todas las rutinas de sincronización mediante un método de *busy–waiting*, que consume ciclos de CPU sin liberarlo hasta que el tiempo de espera haya pasado o que un mensaje sea recibido. Aunque este método desperdicia poder de cómputo, resulta en una sincronización más precisa. Al realizar busy–waiting en RTAI no permitimos que otra tarea utilice el procesador ya que éste nunca se libera, de esta forma la latencia se minimiza. Sin embargo, como el procesador no es liberado, sólo podremos correr un hilo por procesador, lo que limita el número de hilos que pueden correr en paralelo.

RTAI también ofrece al usuario la posibilidad de elegir en qué procesador físico ejecutar un hilo dado, i.e., podemos asignar uno hilo por procesador. Por lo tanto, usando SRTS, podremos dividir el modelo en tantos sub–modelos como procesadores físicos haya disponibles. En nuestra plataforma de hardware de prueba podemos utilizar hasta 12 Procesadores Lógicos (ya que la tecnología hyper–threading divide cada core en dos procesadores virtuales).

Latencia y Overrun

Ante la ausencia de latencia y despreciando el tiempo de proceso de los mensajes, transiciones y otros cálculos, el algoritmo descripto en este capítulo simula correctamente el modelo. En una implementación real sin embargo, no podemos ignorar los efectos de la latencia y a veces los cálculos toman más tiempo que el tiempo físico disponible para ellos, luego la simulación entra en situación de overrun.

Por ejemplo, si el procesador físico necesita 1 segundo de tiempo físico para simular 1 segundo de tiempo lógico, no podremos utilizar un factor de escalado de tiempo real mayor que 1. Si lo hiciéramos el hilo nunca sería capaz de “alcanzar” al tiempo físico escalado. Mientras que este tipo de situaciones de overrun pueden ser evitadas utilizando un factor de escalado chico, algunas situaciones son

irremediables. Por ejemplo, cuando un integrador cuantificado de QSS emite un evento generalmente dispara la evaluación de varias funciones estáticas que deben ocurrir en el mismo instante de tiempo lógico y por lo tanto físico. Aunque el primer evento emitido por el integrador sea correctamente sincronizado, todos los eventos siguientes estarán en overrun.

La latencia también introduce error en la sincronización, que afecta el canal de comunicación. Si un evento que está agendado para t_1 es emitido en el tiempo $t_1 + \Delta t$, los otros hilos lo recibirán con un retardo de Δt. Si durante ese retardo los otros hilos ya calcularon un evento, aparecerá un error que afecta el resultado global.

La plataforma de hardware utilizada muestra en RTAI una latencia promedio en el rango de los 150 hasta 500 $nsec$, con un máximo cercano a los 10 μsec. Estos valores limitan el máximo factor de escalado que podemos utilizar. Por ejemplo, si utilizamos un factor de escalado de 1000, una latencia de 10 μsec será traducida a 10 $msec$ de error en el tiempo de simulación en los eventos transmitidos entre distintos Procesadores Lógicos. Si ese error de sincronización provoca un error numérico inaceptable en el resultado, estaremos obligados a utilizar un factor de escalado más chico.

5.2. Adaptive Scaled Real Time Syncrhonization – ASRTS

Una de las mayores desventajas de SRTS es que el factor de escalado r debe ser elegido por el usuario. A menos que el usuario realice algunos experimentos de prueba y error, estará obligado a saber a priori qué valor utilizar para el factor de escalado. Otro problema es que r se mantiene constante durante toda la simulación. En muchos casos, la carga computacional de la simulación varía con el tiempo, por lo cual tiene sentido adaptar r de acuerdo a la carga actual. En esta sección, introducimos una versión adaptiva de SRTS, llamada ASRTS, donde el factor de escalado de tiempo real es automáticamente adaptado para optimizar la eficiencia de la simulación.

5.2.1. Idea Básica

Adaptive SRTS intenta modificar el valor del factor de escalado automáticamente de forma de minimizar el tiempo que los procesadores desperdician esperando (ya que es tiempo perdido) pero sin provocar situaciones de overrun. De esta manera, ASRTS mejora la eficiencia global bajando el tiempo de simulación.

La idea detrás de ASRTS es muy sencilla. Funciona como SRTS, sólo que cambia el factor de escalado de tiempo real periódicamente. ASRTS divide la simulación en períodos de muestreo de igual duración. Durante cada uno de estos períodos, cada Procesador Lógico lleva cuenta de cuánto tiempo ha pasado esperando. Al finalizar el período, uno de los hilos colecta los tiempos de espera acumulados de todos los Procesadores Lógicos y determina el mínimo (i.e., el tiempo de espera del procesador que tuvo la mayor carga computacional). Usando esta información, calcula la relación entre el menor tiempo de espera acumulado y la longitud del período de muestreo. Si la relación mínima de espera w es mayor que un parámetro de relación de espera deseado w_0 (utilizamos valores cercanos

a 15 %), el algoritmo puede incrementar el factor de escalado de tiempo real y acelerar la simulación. En el caso contrario el algoritmo achica r. De esta manera, ASRTS trata de mantener la relación de espera mínima cercana a la relación de espera deseada w_0.

Los cambios en el factor de escalado de tiempo real son realizados sincrónicamente, esto es, *todos* los Procesadores Lógicos cambian el valor de r en el mismo tiempo (lógico y físico) y al mismo valor, ya que el factor de escalado afecta directamente la sincronización implícita de todo el grupo de Procesadores Lógicos. Una vez que r ha sido cambiado, los Procesadores Lógicos continúan la simulación como si fuera SRTS hasta que el próximo período de sampleo termine.

5.2.2. Algoritmo de Adaptive SRTS

Como mencionamos antes, ASRTS es idéntico a SRTS pero incluye *checkpoints* periódicos (con un período de ΔT) donde el factor de escalado r puede ser cambiado. En cada período de muestreo, ASRTS colecta estadísticas de la carga computacional y del tiempo utilizado en esperar por cada LP. Cuando se alcanza el siguiente checkpoint, los Procesadores Lógicos *detienen* la simulación y esperan hasta que el nuevo factor de escalado de tiempo real haya sido calculado. Este cálculo es realizado por un hilo *coordinador* (el hilo con índice 0), mientras que los hilos restantes esperan en una barrera síncrona que los detiene hasta que r haya sido calculado.

Al final de cada período de sampleo, luego de que cada Procesador Lógico ha calculado su tiempo de espera acumulado T_{w_i}, el coordinador calcula el nuevo factor de escalado como sigue:

1. Busca la relación de espera mínima

$$w = \frac{\text{mín}_i\{T_{w_i}, \sigma\}}{\Delta T}$$

   donde σ es un valor menor pero cercano a 1 para evitar la división por cero.

   En una implementación real no pueden todos los T_{w_i} ser 1 a causa de latencia

2. Calcula el factor *óptimo* de escalado de tiempo real $\hat{r}$ para la relación de espera deseada w_0 como:

$$\hat{r} = r\frac{1 - w_0}{1 - w} \tag{5.1}$$

3. Si $\hat{r} < r$, asigna el nuevo factor de escalada como $r := \hat{r}$, decrementando la velocidad de la simulación.

4. En caso contrario si $\hat{r} \geq r$, calcula el nuevo factor de escalado como:

$$r := \lambda r + (1 - \lambda)\hat{r} \tag{5.2}$$

   donde λ es un parámetro que discutiremos en la Sección 5.2.4. Este ajuste incrementa suavemente la velocidad de acuerdo al *autovalor* discreto [1] λ.

[1] Notar que la Ec.(5.2) es una ecuación en diferencias con λ como autovalor.

Como vemos, el algoritmo de ASRTS tiene 3 parámetros de ajuste: el período de sampleo ΔT, la relación de espera deseada w_0, y el autovalor discreto λ. Discutimos la elección de valores adecuados para estos parámetros más adelante en este capítulo. ASRTS hace uso de la Ec. (5.1) que calcula el factor de escalado óptimo $\hat{r}$, para el cual una relación de espera w_0 es obtenida. Esta expresión es desarrollada en la siguiente sub-sección.

5.2.3. Factor Óptimo de Escalado

Como mencionamos antes, ASRTS intenta llevar la relación de espera mínima w a la relación de espera deseada w_0 ajustando el factor de escalado de tiempo real r.

Supongamos que durante un período de sampleo ΔT el Procesador Lógico con la mayor carga computacional tuvo una relación de espera w cuando el factor de escalado fue r. Durante este período, utilizó en cálculos (no esperando) un total de

$$T_c = \Delta T - w \cdot \Delta T = (1 - w) \cdot \Delta T$$

unidades de tiempo, y la simulación avanzó en $r \cdot \Delta T$ unidades de tiempo.

Luego, para avanzar $r \cdot \Delta T$ unidades de tiempo de simulación, el LP necesitó T_c unidades de tiempo físico. Podemos definir la carga computacional como sigue:

$$W \triangleq \frac{\text{tiempo físico}}{\text{tiempo de simulación}} = \frac{T_c}{r \cdot \Delta T} = \frac{1 - w}{r}$$

Si supusimos que la carga computacional cambia lentamente en relación con ΔT[1], podemos esperar que en el próximo período de sampleo se mantendrá casi sin cambios. Si usamos el factor de escalado de tiempo real *correcto* $\hat{r}$ en el nuevo período de sampleo, deberíamos obtener la relación de espera deseada w_0 y luego,

$$W = \frac{1 - w}{r} = \frac{1 - w_0}{\hat{r}}$$

de donde obtenemos la Ec. (5.1).

5.2.4. Selección de Parámetros

Para seleccionar de forma correcta los parámetros del método de ASRTS, debemos tener en cuenta las siguientes consideraciones:

- El período de sampleo ΔT debe ser grande comparado al tiempo físico necesario para la rutina de re–sincronización y también comparado al tiempo necesario para calcular suficientes eventos. De esta forma el tiempo gastado en la rutina de sincronización es despreciable y cada período ocurre luego de calcular suficientes eventos para que las estadísticas de carga computacional tengan sentido. Por otro lado ΔT debe ser lo suficientemente chico para que el factor de escalado de tiempo real sea modificado varias veces durante la simulación y que sea capaz de reaccionar rápidamente a cambios en la carga computacional.

[1] En caso contrario podemos modificar ΔT.

En la implementación de PowerDEVS utilizamos $\Delta T = 100msec$. En nuestra plataforma, la rutina de sincronización requiere entre 10 a 100 μsec y trataremos con simulaciones que toman varios segundos en finalizar (de lo contrario la paralelización no sería necesaria). Entonces, un período de $\Delta T = 100msec$ es una decisión razonable en la mayoría de los casos.

- La relación de espera deseada w_0 debe ser mayor que 0 y menor que 1. Queremos que la simulación se realice los más rápido posible, minimizando el tiempo de simulación, por lo tanto deberíamos elegir valores pequeños de w_0. Sin embargo, si w_0 es muy cercano a 0 podrían ocurrir muchas situaciones de overrun que causarán un gran error numérico.

  En la implementación de PowerDEVS usamos valores cercanos a $w_0 = 0{,}15$, lo que significa que el factor de escalado de tiempo real es ajustado para que el Procesador Lógico con mayor carga computacional "gaste" aproximadamente 15 % del tiempo en una rutina de espera.

- El autovalor discreto λ (con $0 < \lambda < 1$) determina qué tan rápido se aumenta el factor de escalado para alcanzar el valor óptimo $\hat{r}$ de acuerdo a la Ec.(5.2). Cuando λ tiene un valor cercano a 0 la adaptación es rápida. Cuando tiene un valor cercano a 1 la adaptación es lenta y suave. En presencia de cambios rápidos de la carga computacional una adaptación lenta previene que el factor de escalado tome valores muy grandes que podrían causar overrun.

  Por esta razón utilizamos generalmente $\lambda = 0{,}9$. Con este valor el factor de escalado de tiempo real alcanza el 80 % de su valor final luego de 10 períodos (i.e., 1 segundo en nuestra implementación).

5.3. Relación con otros algoritmos

Las dos técnicas presentadas en esta Tesis tienen relación con dos ideas previas en el mundo de PDES y DEVS. Primero, está relacionado con el RTDEVS executive [45] ya que la simulaciones se ejecutan sincronizadas con el tiempo físico, aunque en nuestro caso, una versión escalada del tiempo físico. Por otro lado el formalismo RTDEVS no es utilizado para la descripción de modelos sino que DEVS. En nuestro caso, las actividades que consumen tiempo son de hecho el cálculo de los eventos de salida y las esperas asociadas a la sincronización.

SRTS y ASRTS están también relacionadas con la técnica de NOTIME [79] ya que la restricción causal sobre los eventos puede ser violada. Estas violaciones de la restricción causal están minimizadas por la introducción de las esperas que realizan los Procesadores Lógicos contra el tiempo físico. Estas esperas actúan como carga computacional "artificial" que ayudan a balancear la carga entre los Procesadores Lógicos que implícitamente resuelven el problema de desbalanceamiento de NOTIME.

5.4. Ejemplos y Resultados

En esta sección presentamos algunos resultados de simulación. Simulamos aquí tres ejemplos de diversos sistemas de gran escala usando los métodos de QSS, primero en forma secuencial y luego utilizando los algoritmos de SRTS y ASRTS.

Todos los ejemplos fueron ejecutados en una PC con un procesador multicore Intel i7 970 – 6 cores con hyper–threading [62] 2GB de RAM utilizando RTAI 3.8, Linux 2.6.32 y PowerDEVS 2.3. RTAI reporta una latencia promedio de 416 $nsec$ y $18,000$ $nsec$ máxima en esta plataforma. Todas las simulaciones fueron repetidas 10 veces y los resultados no exhiben ningún cambio apreciable.

5.4.1. Control de Energía de un Conjunto de Aires Acondicionados

En esta Sección utilizamos el caso de prueba realizado con Vectorial DEVS visto en la Sección 4.6.2.

Simulamos primero el sistema utilizando el método QSS3 con una cuantificación de $\Delta Q_{rel} = 10^{-3}$ y $\Delta Q_{min} = 10^{-6}$ de forma secuencial.

La Figura 5.3 muestra el consumo energético promedio del grupo $P(t)/P_{\text{máx}}$.

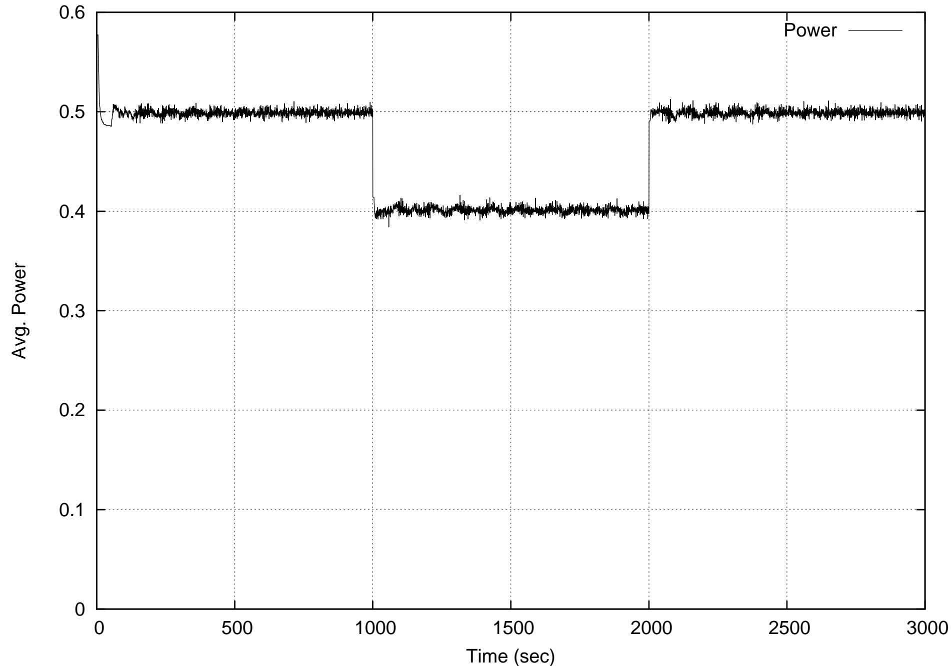

Figura 5.3: Consumo energético – Sistema AA (simulación secuencial).

El tiempo de simulación final utilizado fue $t_f = 3000$ segundos, y la simulación secuencial consumió 58 segundos de tiempo físico.

Luego dividimos el sistema en 12 sub–sistemas con 200 aires acondicionados en cada sub–modelo. El primer modelo incluye también el control PI. Aplicamos la técnica SRTS con distintos factores de escalado de tiempo real. Comenzamos con un factor de escalado de $r = 200$, el cual consume $t_f z/r = 3000/200 = 50$ segundos de tiempo físico. Obtuvimos buenos resultados utilizando factores de escalado hasta $r = 950$ que consume $t_f/r = 3000/950 = 3{,}16$ segundos. Para cada factor de escalado comparamos el error de simulación contra la versión secuencial y medimos el error relativo RMS. Tanto aquí como en los siguientes ejemplos el

error RMS es computado como:

$$RMS = \frac{\text{mean}(x - \hat{x})^2}{\text{mean}(x)}$$

donde x es el resultado secuencial y $\hat{x}$ es el resultado paralelo.

La Figura 5.4 muestra el resultado del error en función del factor de escalado de tiempo real.

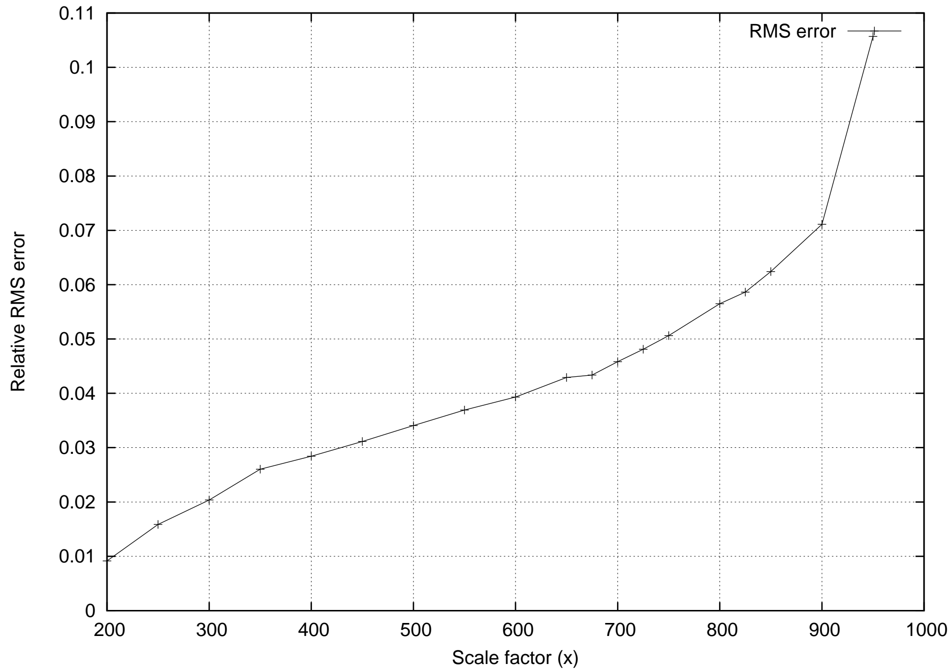

Figura 5.4: SRTS: Error *vs* factor de escalado de tiempo real – Ejemplo AA.

En este ejemplo utilizando $r = 950$, SRTS es 18 veces más rápido que la simulación secuencial sin introducir un error numérico inaceptablemente grande.

Sin embargo para $r > 500$ observamos que los Procesadores Lógicos estaban la mayor parte del tiempo en overrun. De todas formas, en este caso el overrun no afecta mucho al error numérico.

Luego simulamos el sistema utilizando Adaptive SRTS con $\Delta T = 100$ msec, $w_0 = 0{,}15$, y $\lambda = 0{,}9$. La simulación con ASRTS tomó 6,6 segundos, alcanzando un valor máximo para el factor de escalado de tiempo real de $r = 480$ (con un error de 0,03)y sin requerir experimentación previa para conocer el factor de escalado de tiempo real.

Por lo tanto, ASRTS fue casi 9 veces más rápido que la simulación secuencial utilizando 12 procesadores. El factor de escalado de tiempo real alcanzado fue cercano al límite en el cual se observa overrun.

Repetimos las simulaciones para distintos valores de λ. La Figura 5.5 muestra la forma en que ASRTS adapta el factor de escalado durante cada simulación ejecutada.

Se puede ver que r converge muy lento al factor de escalado óptimo para valores

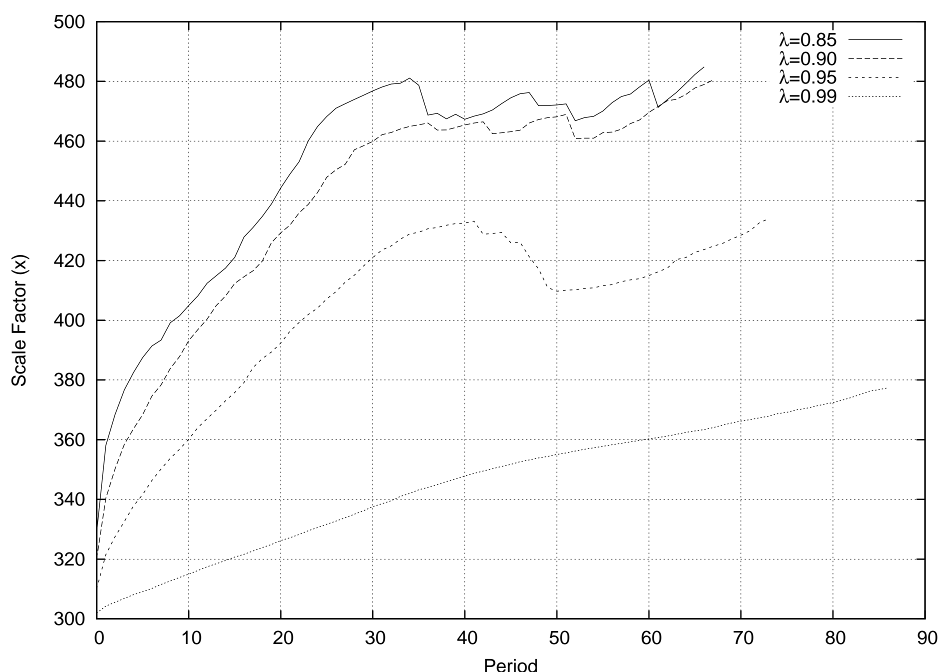

Figura 5.5: ASRTS: Adaptación del Factor de Escalado de Tiempo Real para distintos λ – Ejemplo Aires Acondicionados.

grandes de λ (cercanos a 0,99) por lo cual la simulación es lenta. Por otro lado r converge rápido para valores pequeños de λ (cercanos a 0,8) pero requiere reducir su valor frecuentemente. Una reducción del factor de escalado de tiempo real indica que algún Procesador Lógico esperó menos que la relación de espera desead w_0 lo cual es inseguro ya que podría indicar que hubo overrun. Un valor de $\lambda \approx 0{,}9$ provee un buen compromiso entre una adaptación rápida y evitar situaciones de overrun.

Finalmente repetimos las simulaciones con ASRTS variando el número de procesadores, dividiendo el model en 2, 4, 6, 8 y 12 sub–modelos. La Figura 5.6 muestra la aceleración obtenida para estos casos.

Podemos observar una evolución casi lineal de la aceleración con el número de procesadores, lo cual es deseable cuando se paraleliza una simulación.

5.4.2. Cadena de Inversores Lógicos

Un inversor lógico es un dispositivo que realiza una operación lógica en una señal. Cuando la señal de entrada toma un valor alto, la salida del inversor toma un valor bajo y vice versa.

Los inversores lógicos son implementados por circuitos eléctricos por lo cual muestran una respuesta no ideal debido a que el tiempo de ascenso y descenso de la señal de salida está limitada por características físicas y el nivel de salida no es alcanzado inmediatamente, i.e., es retardado.

Una cadena de inversores es una concatenación de muchos inversores donde la salida de cada inversor actúa como la entrada al próximo. Haciendo uso de las

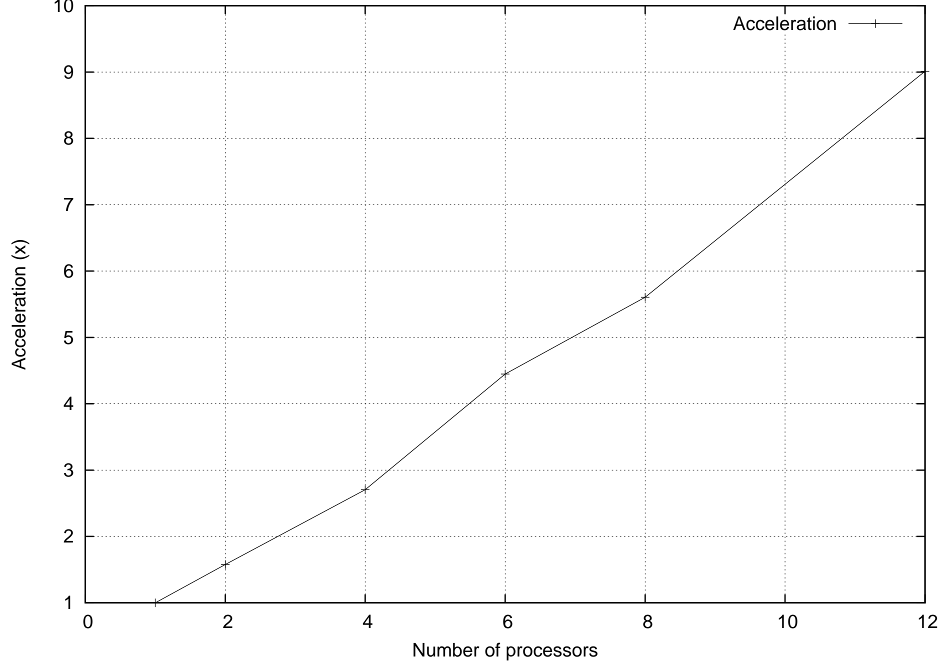

Figura 5.6: ASRTS: Aceleración *vs* número de procesadores – Ejemplo Aire Acondicionados.

limitaciones físicas antes mencionadas, una cadena de inversores puede ser utilizada para obtener señales retardadas. Consideramos aquí una cada de m inversores de acuerdo al modelo dado por Savcenco y Mattheij [80] el cual está caracterizado por las siguientes ecuaciones:

$$\begin{cases} \dot{\omega}_1(t) &= U_{op} - \omega_1(t) - \Upsilon g\left(u_{in}(t), \omega_1(t)\right) \\ \dot{\omega}_j(t) &= U_{op} - \omega_j(t) - \Upsilon g\left(\omega_{j-1}(t), \omega_j(t)\right) \quad j = 2, 3, .., m \end{cases} \tag{5.3}$$

donde:

$$g(u, v) = \left(\text{máx}(u - U_{thres}, 0)\right)^2 - \left(\text{máx}\left(u - v - U_{thres}, 0\right)\right)^2 \tag{5.4}$$

Utilizamos el juego de parámetros dado en el artículo: $\Upsilon = 100$ (por el cual el sistema resulta muy stiff), $U_{thres} = 1$, y $U_{op} = 5$. Las condiciones iniciales son, como en el artículo original, $\omega_j = 6{,}247 \cdot 10^{-3}$ para valores impares de j y $\omega_j = 5$ para valores pares de j. La entrada es una señal trapezoidal periódica con parámetros $V_{up} = 5V$, $V_{low} = 0V$, $T_{down} = 10$, $T_{up} = 5$, $T_{rise} = 5$, y $T_{fall} = 2$.

En este caso consideramos un sistema con $m = 504$ inversores, lo que resulta en 504 ecuaciones diferenciales con 1008 condiciones de discontinuidad debido a la función 'máx' en la Ecuación (5.4).

Como en el ejemplo anterior, simulamos primero el sistema de forma secuencial utilizando el método apto para sistemas rígidos LIQSS3, que ha sido agregado a PowerDEVS [64]. El resultado del voltaje en el último inversor se muestra en la

Figura 5.7.

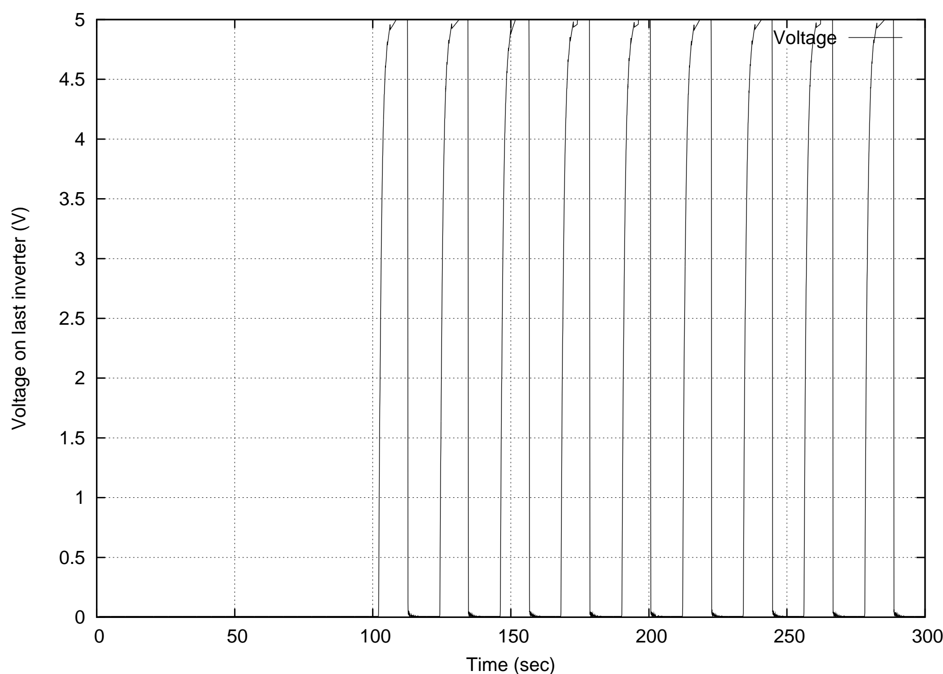

Figura 5.7: Voltaje en el último inversor(simulación secuencial).

Luego dividimos el sistema en 12 sub–modelos con 42 inversores cada uno y lo simulamos con SRTS y distintos factores de escalado de tiempo real. En este caso obtuvimos buenos resultados con r hasta $r \approx 35$.

La Figura 5.8 muestra el error en función del factor de escalado de tiempo real. Aquí el error RMS luce grande debido a un pequeño corrimiento temporal en el voltaje de salida. Como la señal muestra cambios abruptos, un pequeño corrimiento temporal puede causar un gran error RMS.

Luego aplicamos el algoritmo ASRTS utilizando 12 procesadores y encontramos que la simulación es 4,5 veces más rápida que la secuencial.

En este ejemplo, la carga no está distribuida equitativamente entre los sub–sistemas. A medida que la onda viaja, provoca una gran carga computacional en el procesador que calcula las etapas que están cambiando. En contraste, los procesadores que computan estados que no están cambiando muestran una carga computacional muy baja. Por lo tanto en este ejemplo ni SRTS ni ASRTS pueden explotar la paralelización tan eficientemente como en el ejemplo anterior.

Repetimos las simulaciones utilizando ASRTS variando el número de procesadores. La Figura 5.9 muestra la aceleración obtenida en cada caso.

Aquí de nuevo, la aceleración obtenida se incrementa casi linealmente con el número de procesadores en uso.

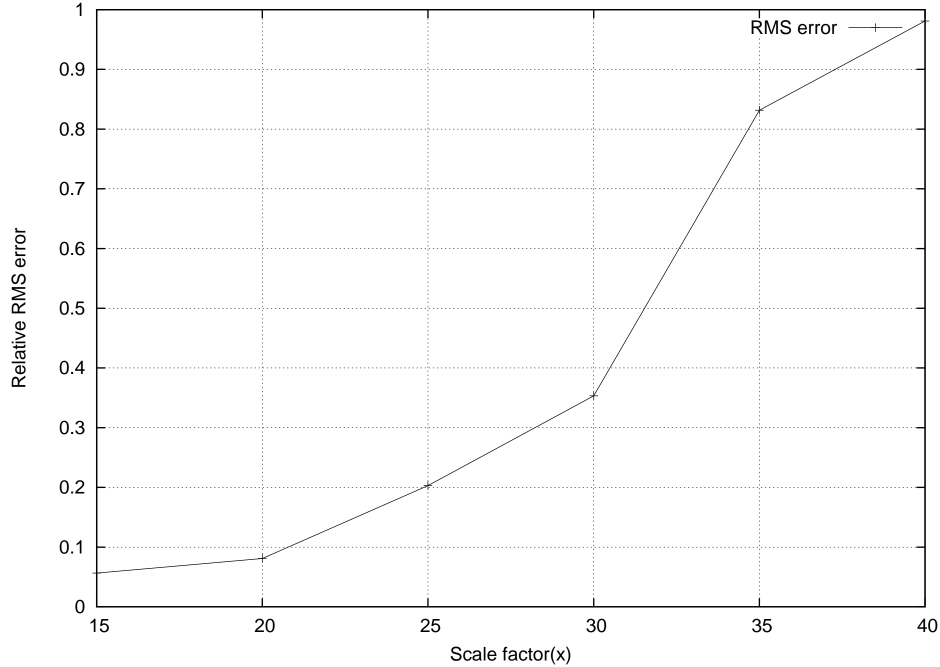

Figura 5.8: SRTS: Error *vs* factor de escalado de tiempo real – Ejemplo cadena de inversores.

5.4.3. Línea de Transmisión LC

Consideramos ahora el sistema visto en la Sección 4.6.1 sólo que ahora la entrada es una señal senoidal:

$$u_0(t) = \sin(\omega t) \tag{5.5}$$

con $w_0 = 0{,}13$ Hz.

Como en los casos anteriores primero simulamos el sistema de manera secuencial utilizando el método QSS3. La Figura 5.10 muestra el voltaje resultante al final de la línea de transmisión $u_{600}(t)$.

Luego, dividimos el sistema en 12 sub–modelos con 50 segmentos en cada uno y lo simulamos con SRTS utilizando distintos valores para el factor de escalado de tiempo real. La Figura 5.11 muestra el error en función del factor de escalado.

En este caso obtuvimos buenos resultados para $r < 400$. Para valores más grandes del factor de escalado, la simulación se vuelve numéricamente inestable rápidamente. Esto es de esperar ya que la adición de retardos en un sistema marginalmente estable tiende a causar inestabilidad.

Luego simulamos el sistema con ASRTS. Con 12 procesadores ASRTS pudo acelerar el sistema ejecutándolo 5,5 veces más rápido que la simulación secuencial.

Como en el ejemplo anterior, la carga no está balanceada equitativamente mientras la onda viaja desde la entrada a la salida, lo que limita la ganancia de la paralelización.

Luego analizamos el uso de ASRTS variando el número de procesadores utilizados obteniendo de nuevo una relación cercana a lineal entre el número de

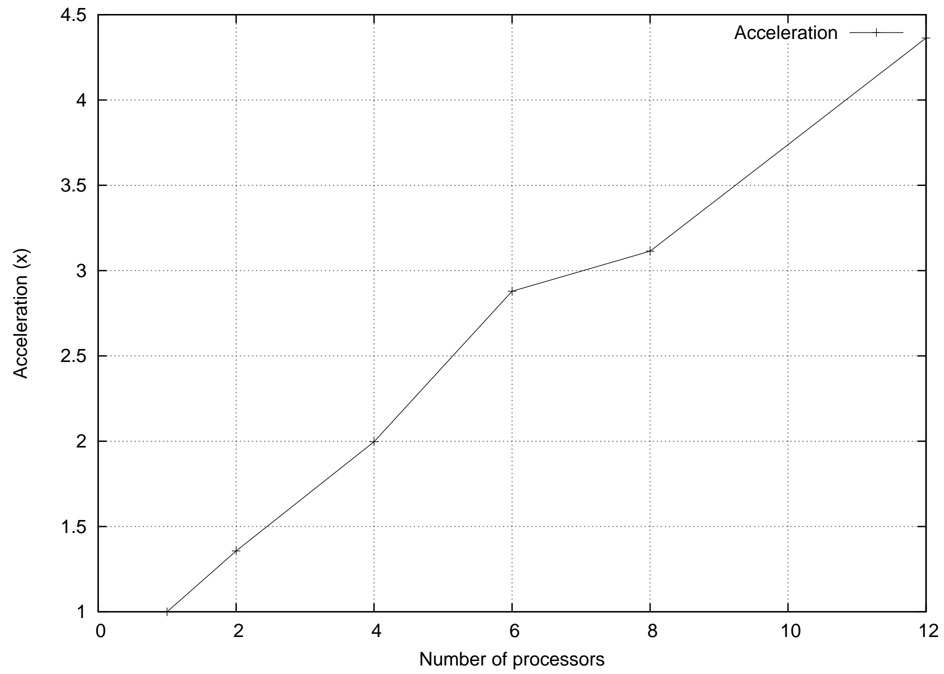

Figura 5.9: ASRTS: Aceleración *vs* número de procesadores – Ejemplo cadena de inversores.

procesadores en uso y la aceleración como lo muestra la Figura 5.12.

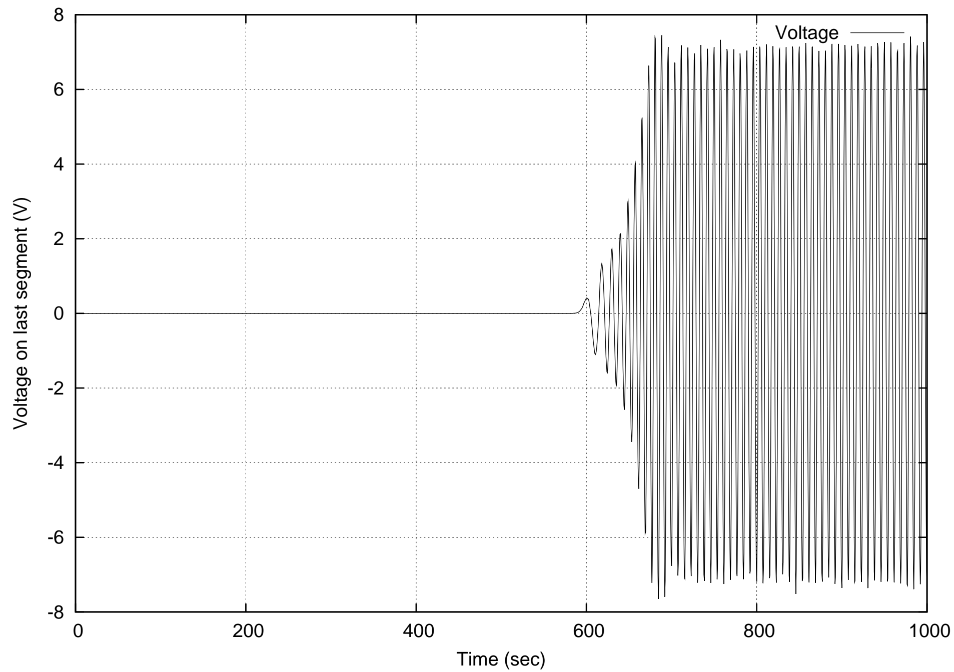

Figura 5.10: Voltaje en el último segmento – Ejemplo Línea LC (simulación secuencial).

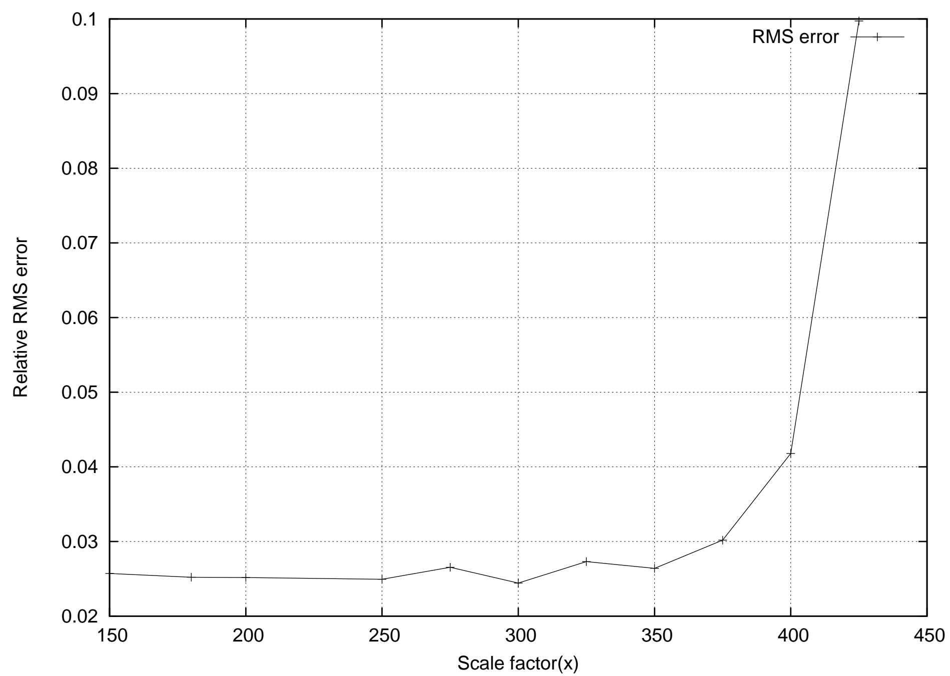

Figura 5.11: SRTS: Error RMS *vs* factor de escalado – Ejemplo Línea LC.

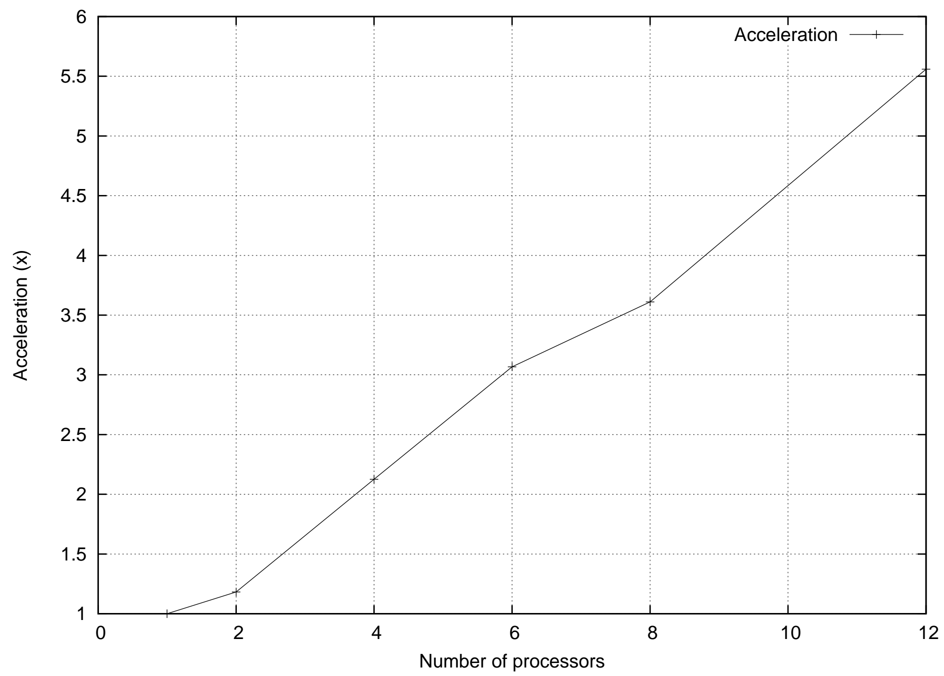

Figura 5.12: ASRTS: Aceleración *vs* Número de Procesadores – Ejemplo Línea LC.

5.5. Conclusiones

En este capítulo introdujimos SRTS y ASRTS, dos novedosas técnicas para la simulación de sistemas continuos e híbridos en paralelo mediante eventos discretos.

Ambas técnicas están basadas en la idea de sincronizar el tiempo de simulación de cada Procesador Lógico con una versión escalada del tiempo físico. Como todos los Procesadores Lógicos están sincronizados con el tiempo físico, están indirectamente sincronizados entre ellos. Esta sincronización indirecta permite ejecutar cada proceso independientemente de los otros evitando los costos de la sincronización y coordinación entre procesos.

Implementamos estas técnicas en PowerDEVS utilizando el sistema operativo de tiempo real RTAI y el entorno desarrollado en el Capítulo 3. El RTOS fue necesario para asegurar una sincronización precisa, para asignar hilos con CPU y para evitar que el OS (Linux) quite el procesador a un hilo que esté ejecutando.

Analizamos la aplicación de estas técnicas a tres ejemplos y estudiamos su performance y limitaciones. Todos los ejemplos de este capítulo fueron descriptos utilizando modelos Vectorial DEVS introducidas en el Capítulo 4. Utilizando 12 procesadores, encontramos grandes mejoras en el tiempo de simulación. En los ejemplos analizados, ASRTS los simuló desde 4,5 a 9 veces más rápido que la simulación secuencial. La versión no–adaptiva $SRTS$ es capaz de simularlos un poco más rápido pero requiere experimentación previa para determinar el valor adecuado del factor de escalado de tiempo real.

También analizamos el error numérico adicional introducido por estas técnicas. Encontramos que si el factor de escalado de tiempo real r se mantiene lo suficientemente chico para evitar overruns, el error se mantiene acotado. Además ASRTS ajusta r para evitar overruns por lo cual el error introducido por ASRTS es pequeño.

Todo el código de la implementación de SRTS y ASRTS en PowerDEVS puede ser descargado del branch *real–time* del SVN [77].

Capítulo 6

Simulación de Modelos Modelica

Modelica [34, 35] es un lenguaje orientado a objetos basados en ecuaciones que permite representar tanto sistemas continuos como híbridos usando un conjunto de ecuaciones no-causales. El lenguaje Modelica provee una forma estandarizada para modelar sistemas físicos complejos multidominio que contienen por ejemplo sub–componentes mecánicos, eléctricos, electrónicos, hidráulicos, térmicos, de control, de energía eléctrica u orientados a procesos.

Existen herramientas comerciales tales como Dymola [74] y Scicos [70] junto con implementaciones OpenSource como OpenModelica [33] que permiten el modelado y simulación de modelos descriptos en el lenguaje Modelica. Todas estas herramientas realizan una serie de pasos de pre–procesamiento (aplanamiento del modelo, reducción de índice, ordenamiento y optimización de las ecuaciones) y convierten el modelo en un conjunto explícito de Ecuaciones Diferenciales Ordinarias (ODE del inglés) de la forma $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$. Luego, se genera código C++ eficiente que realiza la simulación. Estas herramientas también poseen algoritmos de integración (o solvers) numéricos para ODEs que evalúan el lado derecho de la ODE en pasos de tiempo discretos t_k para computar el próximo valor del vector de estado $\mathbf{x_{k+1}}$. Por lo tanto, estos entornos de simulación utilizan generalmente **time slicing**, i.e. sus algoritmos de simulación están basados en la discretización temporal en vez de cuantificación de estados.

En este Capítulo estudiamos diversas técnicas que utilizan los métodos de cuantificación de estado aplicado a modelos descriptos en el lenguaje Modelica y presentamos varios ejemplos de uso de las mismas.

6.1. Simulación de modelos Modelica en PowerDEVS

Los **métodos de QSS** y el principio de **cuantificación de estado** son prometedores para simular cierta clases de problemas del mundo real. Para simular un sistema con los métodos de QSS utilizando PowerDEVS [11] el usuario necesita un conocimiento extenso del formalismo DEVS. Más específicamente, el modelo debe ser convertido primero manualmente en una formulación ODE explícita, las de-

pendencias entre sus–sistemas deben ser identificados y la estructura DEVS debe ser descripta. Incluso si un usuario posee el conocimiento necesario para realizar estos pasos, este enfoque es sólo viable para sistemas pequeños.

PowerDEVS no soporta el modelado orientado a objetos mientras que Modelica sí. Por estas razones es mucho más conveniente para el usuario describir los modelos en el lenguaje Modelica que en PowerDEVS.

En esta sección se presenta una metodología para unificar el potente lenguaje orientado a objetos Modelica por un lado, con la plataforma de simulación de PowerDEVS basada en QSS. En [30] una primera versión de la interfaz entre OpenModelica y PowerDEVS para sistemas sin discontinuidades fue presentada y analizada. Extendemos aquí esta interfaz para incluir modelos con discontinuidades y achicar el espacio que hay entre estas dos herramientas.

6.1.1. Simulación de Modelos Discontinuos de Modelica utilizando Métodos de QSS

Esta Sección describe una forma de simular un modelo Modelica usando los métodos de QSS vistos en la Sección 2.2. Por simplicidad, supusimos que el modelo está descripto por una ODE, pero la interfaz es capaz de simular sistemas DAE también. Escribamos la Ec. 2.15 expandida a sus componentes individuales olvidándonos temporalmente de la parte discontinua:

$$\begin{aligned} \dot{x}_1 &= f_1(q_1, \ldots, q_n, t) \\ &\vdots \\ \dot{x}_n &= f_n(q_1, \ldots, q_n, t) \end{aligned} \tag{6.1}$$

Si consideramos una componente de la Ec. 6.1, podemos dividirla en dos ecuaciones:

$$q_i = Q(x_i) = Q(\int \dot{x}_i \; dt) \tag{6.2}$$

$$\dot{x}_i = f_i(q_1, \ldots, q_n, t) \tag{6.3}$$

Como vimos previamente, estas ecuaciones pueden ser representadas por un acoplamiento de modelos DEVS que las aproximan utilizando los métodos de QSS (ver Figura 2.8).

Nos queda lidiar con las discontinuidades que puede presentar el modelo Modelica. Debemos representar las discontinuidades de Modelica como bloques DEVS que usen la aproximación de QSS.

Describiendo Discontinuidades en Modelica

Las discontinuidades en sistemas dinámicos están relacionadas directamente a la noción de eventos. Podemos distinguir dos tipos de eventos: eventos temporales y eventos de estado.

Eventos Temporales

Los eventos temporales corresponden a cambios de estado como una función de la variable interna *time* que evoluciona continuamente. Estos eventos pueden ser agendados antes de iniciar la simulación ya que es posible predecir el momento en el tiempo cuando ocurrirán. Los eventos temporales en Modelica pueden estar descriptos de dos formas [32]:

- Con una expresión condicional de tiempo discreto que contiene la variable *time* (por ejemplo en sentencias when) de la forma:
  time >= expresión de tiempo discreto, por ejemplo $t >= t_e$

- Con una sentencia periódica *sample* de la forma:
  sample(inicio, intervalo) que genera eventos en tiempos pre–definidos.

El primer caso puede ser tratado formulando una función de cruce por cero de la forma:

$$g(t) = t - t_e$$

Cuando g(t) cruza por cero, un evento debe ser generado. Veremos luego qué bloques DEVS deben ser definidos para generar esos eventos. La sentencia *sample()* puede ser tratada fácilmente agregando un modelo DEVS atómico que provoque los eventos en los instantes de tiempo pre–definidos.

Eventos de Estado

Los eventos de estado están relacionados con cambios discretos en las variables de estado durante la simulación como una función de otras variables de estado que alcanzan algún valor de umbral. Por lo tanto *no pueden* ser agendados previamente. Un evento de estado puede ser descripto mediante sentencias *when* o *if-then-else* involucrando una o más variables de estado.

Cuando un modelo es compilado por OpenModelica o por Dymola, los eventos de estado son traducidos en **funciones de cruce por cero** de la forma $g_i(\mathbf{x}, t)$. Durante la ejecución de la simulación las funciones de cruce por cero son monitoreadas constantemente y cuando una de ellas cruza por cero, la discontinuidad es detectada y tratada. Por lo tanto, podemos utilizar directamente las funciones de cruce por cero generadas por OpenModelica para identificar los eventos de estado de la misma forma que lo hacemos con los eventos temporales. Lo único que necesitamos es un bloque de **Función Estática** que evalúe la función de cruce y otro bloque de **Detección de Cruce Por Cero** que detecte cuando ocurre un cruce por cero.

Estructura DEVS

El formalismo DEVS [95] permite describir tanto la parte continua como la parte discontinua del modelo a través de un acoplamiento de modelos atómicos DEVS más simples. Específicamente, debemos definir:

- Un bloque de **Integrador Cuantificado** (Ec. 6.2) que tome como entrada la derivada $\dot{x}_i$ y emita como salida q_i.

- Un bloque de **Función Estática** que recibe una secuencia de eventos $q_1, \ldots, q_n$, y calcula la secuencia de valores de derivadas de estados $\dot{x}_i$ (Ec. 6.3). El mismo bloque puede ser utilizado para la evaluación de las funciones de cruce por cero $g_i(\cdot)$

- Un bloque de **Detección de Cruce por Cero** que recibe como entrada la función de cruce evaluada y genere un evento de salida cunado su entrada cruza por cero.

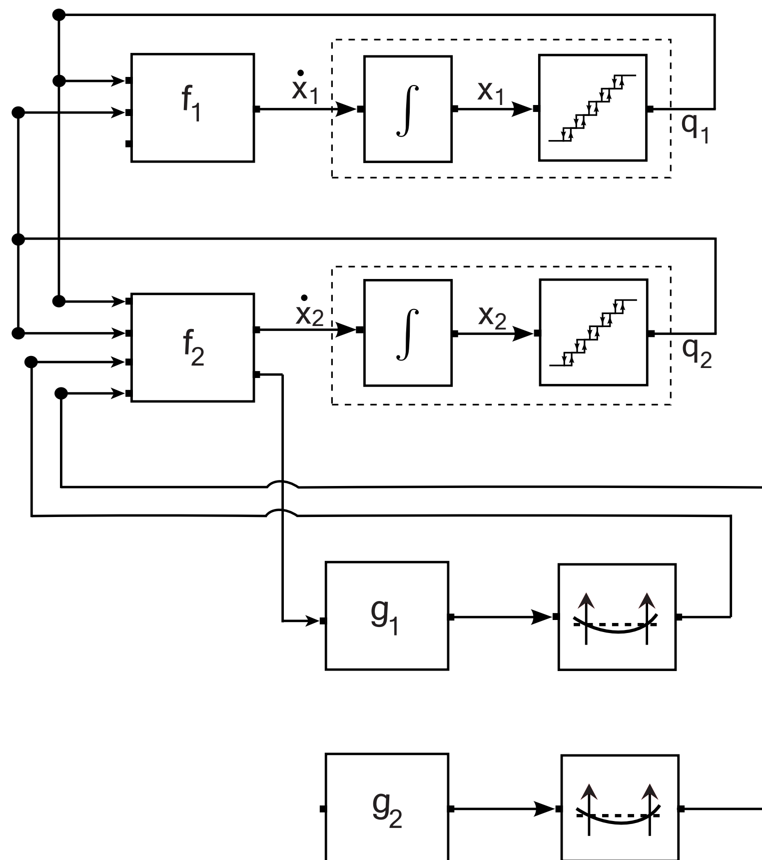

Figura 6.1: Modelo DEVS acoplado para la simulación QSS de un modelo discontinuo con 2 estados y 2 funciones de cruce $g_1(\cdot)$ and $g_2(\cdot)$.

De este modo, podemos simular un modelo discontinuo escrito en Modelica utilizando un modelo DEVS acoplado que contenga los bloques descriptos previamente.

Un diagrama en bloques que representa el modelo DEVS final para un sistema de ejemplo con dos variables de estado (los dos integradores) y dos funciones de cruce ($g1$ y $g2$) puede verse en la Figura 6.1.

6.1.2. Interfaz OpenModelica – PowerDEVS (OMPD)

Esta sección describe el trabajo realizado para simular automáticamente modelos Modelica en PowerDEVS utilizando los algoritmos de QSS.

¿Qué Necesita PowerDEVS ?

Concentrémonos primero en qué requiere PowerDEVS para realizar la simulación de un modelo de Modelica. Como lo muestra la Figura 6.1, un componente esencial de una simulación de PowerDEVS es la estructura gráfica. En PowerDEVS, un modelo puede ser descripto a través de una descripción gráfica (archivos `.pdm`) o mediante archivos de texto (archivos `.pds`). La estructura en nuestra implementación será dada en una forma textual en un archivo de estructura `.pds` que contendrá información acerca de los bloques (los nodos) de la Figura 6.1 junto con información acerca de las conexiones (vértices) entre estos bloques.

Más específicamente, necesitamos incluir en la estructura:

- Un **Integrador Cuantificado** por cada variable de estado con $\dot{x}_i$ como entrada y q_i como salida.
- Un bloque de **Función Estática** para cada variable de estado que recibe como entrada la secuencia de eventos $q_1, \ldots, q_n$, y computa $\dot{x}_i = f_i(\mathbf{q})$.
- Un bloque de **Función Estática** para cada función de cruce por cero $g_i(\cdot)$ generadas por OpenModelica que recibe como entrada las variables sobre las cual depende $g_i(\cdot)$ y evalúa la función emitiendo el valor por su puerto de salida.
- Un bloque de **Detección de Cruce por Cero** detrás de cada bloque de funciones estáticas de cruce por cero. El bloque de detección de cruces emite un evento cada vez que detecta que la función en su entrada cruzó por cero.
- Una **conexión** es agregada si y sólo si existe una dependencia entre dos bloques (uno emite lo que el otro toma como entrada).

Habiendo identificado la estructura DEVS necesaria, debemos ahora especificar qué necesita ser calculado dentro de cada uno de los bloques de funciones estáticas. Los distintos bloques necesitan tener acceso a distintas partes de información.

En la implementación actual, se genera un archivo de código **.cpp** que contiene el código y parámetros para todos los bloques de la estructura. El archivo de código generado contiene la siguiente información:

- Para cada bloque **Integrador Cuantificado** el archivo contiene la condición inicial, tolerancia de error y método de integración (QSS, QSS2, QSS3) para el integrador.
- Para cada bloque de **Función Estática**, contiene las ecuaciones necesarias para computar la derivada de cada variable de estado. Además, la tolerancia de error junto con las entradas y salidas necesarias por el bloque. Si el bloque de función estática representa una función de cruce por cero, el bloque evalúa la función respectiva $g_i(\cdot)$.

¿Qué provee OpenModelica?

En la sección 6.1.2 describimos qué necesita PowerDEVS para realizar la simulación. Nuestro trabajo se focaliza en una forma automática de simular modelos Modelica utilizando métodos de QSS en PowerDEVS. Por lo tanto, los archivos de simulación que necesita PowerDEVS deberían ser generados automáticamente utilizando la información contenida en el modelo Modelica como entrada. Afortunadamente, los software existentes para compilar modelos Modelica como Dymola y OpenModelica producen el código de simulación que contiene toda la información que PowerDEVS necesita. Por lo tanto, podemos hacer uso de una herramienta existente de Modelica modificando el módulo de generación del código en la última etapa del compilador para producir los archivos necesarios por PowerDEVS.

Este trabajo se basa en modificar el Compilador de OpenModelica (OMC) ya que es un software de código abierto y posee una comunidad de desarrolladores en crecimiento constante. OMC toma como entrada un archivo fuente que representa un modelo Modelica y lo traduce primero a un modelo plano, es decir sin clases. El aplanamiento consiste en el análisis del código fuente, chequeo de tipos, expandir las clases heredadas y sus modificaciones, etc. El modelo plano incluye un conjunto de declaraciones de ecuaciones y funciones **sin toda la estructura de objetos**. Luego, se realiza la reducción de índice en el conjunto de ecuaciones para eliminar las dependencias algebraicas estructurales entre las variables de estado. Las ecuaciones resultantes son analizadas, y guardadas en la forma Triangular Inferior a Bloques (o BLT del inglés) y optimizadas. Finalmente, el generador de código de OMC produce el código C++ que es luego compilado. El ejecutable resultante es usado para la simulación del modelo.

La información que se requiere extraer de OMC está almacenada principalmente en la estructura DLOW[1] donde las siguientes partes de información están definidas:

- Ecuaciones: $E = \{e_1, e_2, \ldots, e_N\}$.
- Variables: $V = \{v_1, v_2, \ldots, v_N\} = V_S \bigcup V_R$
  donde V_S es el conjunto de variables de estado con $|V_S| = N_S \leq N$ y V_R es el conjunto de todas las otras variables en el modelo.
- Los bloques BLTs: son subconjuntos de ecuaciones $\{e_i\}$ que deben ser resueltas juntas debido a que son parte de un lazo algebraico.
- Las funciones de cruce por cero: $G = \{g_1, g_2, \ldots, g_K\}$.
- La matriz de incidencia: A_n que indica qué variables son utilizadas en cada ecuación

La interfaz OMPD utiliza la información descripta previamente y realiza los siguientes pasos:

1. **Particionar de las ecuaciones**: La interfaz identifica las ecuaciones necesarias para computar la derivada del estado $\dot{x}_i = f_i(\mathbf{q})$ para cada una de las variables de estado. Las ecuaciones ya divididas son asignadas a bloques DEVS de funciones estáticas correspondiente a la derivada de variable que computan.

[1]DLOW es una estructura interna utilizada por OMC

2. **Mapear las ecuaciones a bloques BLT**: Las ecuaciones divididas son mapeadas de nuevo a bloques BLT para generar código de simulación que resuelva lazos algebraicos lineales/no-lineales si fuese necesario.

3. **Identificar las funciones de cruce por cero**: Las funciones de cruce por cero generadas por OMC son extraídas y asignadas a bloques de funciones estáticas separados.

4. **Construir la matriz de incidencia generalizada** : La matriz de incidencia de $N \times N$ debe ser extendida para incluir también las funciones de cruce por cero y las variables involucradas en éstas. Por lo tanto, debe ser extendida agregando K filas correspondiente a las K funciones de cruce por cero. El resultado es una matriz $K \times N$ de incidencia generalizada.

5. **Generar la estructura DEVS**: Para generar correctamente la estructura DEVS del modelo la dependencia entre los bloques debe ser analizada. Esto se realiza utilizando la matriz de incidencia generalizada para encontrar las entradas y salidas para cada bloque.

6. **Generar el archivo de estructura .pds**: Una vez que se ha calculado la dependencia entre los bloques se conoce la estructura del modelo DEVS y se puede generar entonces el archivo ".pds" de estructura de PowerDEVS trivialmente.

7. **Generar el código para los bloques estáticos**: En este paso la funcionalidad de cada bloque estático es definida a través de código de simulación. Cada bloque estático necesita conocer sus entradas y salidas, identificadas en la estructura DEVS, junto con los bloques BLT necesarios para calcular la derivada del estado correspondiente. Los bloques estáticos responsables de las discontinuidades contienen las funciones de cruce $g_i(\cdot)$ generadas por OMC. El módulo de generación de código de OMC es utilizado para emitir el código de simulación para cada bloque estático que ya ha sido optimizado para resolver lazos algebraicos lineales y no lineales.

8. **Generar el archivo de simulación .cpp**: El código para los bloques estáticos es escrito en el archivo .cpp junto con otra información necesaria (tolerancia de error, tamaño del sistema, etc).

6.1.3. Resultados

En esta Sección presentamos y discutimos los resultados obtenidos utilizando la interfaz OMPD. El objetivo es comparar la eficiencia y precisión de los métodos de QSS contra otros algoritmos de simulación. Específicamente queremos comparar los métodos de QSS2 y QSS3 de PowerDEVS contra los solvers DASSL, Radau IIa y Dopri45 implementados en Dymola v7.4 y la versión de DASSL de OpenModelica v1.5.1.

DASSL fue elegido ya que representa el solver preferido en el estado del arte debido a que es multipropósito, apto para sistemas stiff DAE y es usado por la mayoría de las herramientas de simulación. Radau IIa fue incluido en la comparación debido a que es un algoritmo monopaso (Runge–Kutta) supuestamente es más eficiente que un algoritmo multi-paso (BDF) para sistemas con discontinuidades frecuentes, ya que el control de paso es mucho más costoso para los últimos [21].

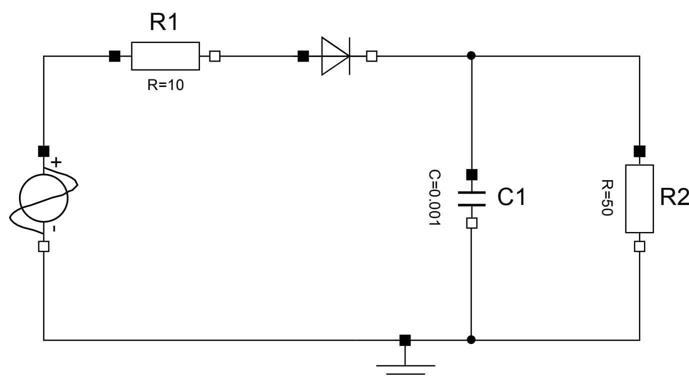

Figura 6.2: Representación gráfica del rectificador de media onda.

Finalmente, Dopri45 fue elegido porqué es un método de Runge-Kutta explícito a diferencia de DASSL y Radau IIa, los cuales son algoritmos implícitos que podrían ser peores a la hora de simular sistemas no-stiff.

Casos de Estudio

Como problemas de estudio elegimos dos problemas del mundo real que exhiben discontinuidades frecuentes. Primero un circuito rectificador de media onda realizado gráficamente con componentes estándar de Modelica como lo muestra la Figura 6.2 y segundo una fuente conmutada como vemos en la Figura 6.3.

Para medir el tiempo de ejecución de cada algoritmo de simulación utilizamos el tiempo reportado por cada herramienta. Dymola informa el tiempo de CPU de la integración, OpenModelica reporta la variable `timeSimulation` y PowerDEVS muestra el tiempo de simulación transcurrido. Para registrar sólo el tiempo de simulación se anuló la generación de archivos de salida en todos los casos. Las simulaciones fueron ejecutadas en una máquina de escritorio Dell 32bit con un procesador Quad-Core @ 2.66 GHz y 4GB de RAM.

El tiempo de CPU medido no debe ser considerado como un valor absoluto ya que variará de una computadora a otra, pero el orden relativo entre los algoritmos debería mantenerse igual.

El cálculo de la precisión de las simulaciones sólo puede ser realizado aproximadamente ya que las trayectorias de los estados en los dos modelos no pueden ser calculadas analíticamente. Para estimar la precisión de los algoritmos de simulación debemos obtener trayectorias de referencia $(\mathbf{t^{ref}}, \mathbf{y^{ref}})$. Para ello utilizamos Dymola con el método por defecto DASSL con una tolerancia baja de 10^{-12} y generando 10^5 puntos de salida equi-espaciados. Además para dar más soporte a la solución de referencia, también se generó una trayectoria de referencia utilizando QSS3 en PowerDEVS también con una tolerancia de 10^{-12}. De todas formas sólo informamos el error de simulación contra la trayectoria de Dymola ya que la diferencia entre las dos soluciones de referencia es del orden de 10^{-6}.

Para computar el error de simulación cada una de las trayectorias simuladas fueron comparadas contra las dos soluciones de referencias. Con este fin forzamos a todos los solvers a emitir 10^5 puntos equi-espaciados obteniendo las trayectorias de simulación $(\mathbf{t^{ref}}, \mathbf{y^{sim}})$ sin cambiar el paso de integración. El error medio absoluto

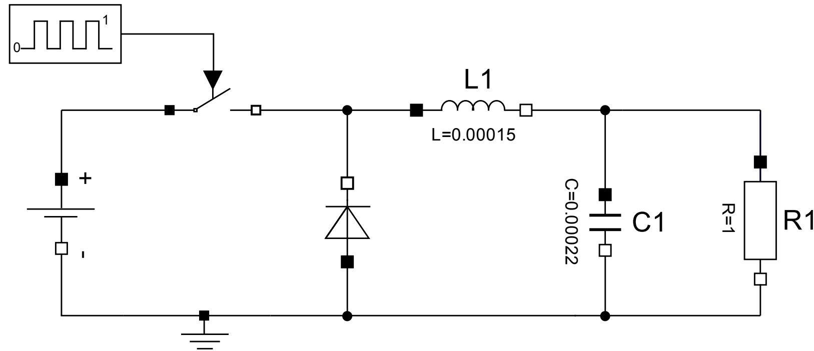

Figura 6.3: Representación gráfica de la fuente conmutada.

es calculado como:

$$error = \frac{1}{|t^{ref}|} \sum_{i=1}^{|t^{ref}|} |y_i^{sim} - y_i^{ref}| \tag{6.4}$$

En el caso que haya más de una variable de estado, reportamos el error medio sobre todas las variables de estado.

Tabla 6.1: Esta tabla muestra los resultados de simulación de varios algoritmos para el rectificador de media onda para 1 seg. de simulación. La comparación realizada incluye el tiempo de CPU (en seg.) junto con la precisión de simulación relativa medida contra la trayectoria de referencia de Dymola.

			Tiempo de CPU (seg)	**Error de Simulación**
Dymola	**DASSL**	10^{-3}	0.019	1.45E-03
	DASSL	10^{-4}	0.022	2.35E-04
	Radau IIa	10^{-7}	0.031	2.20E-06
	Dopri45	10^{-4}	0.024	4.65E-05
PowerDEVS	**QSS3**	10^{-3}	0.014	2.59E-04
	QSS3	10^{-4}	0.026	2.23E-05
	QSS3	10^{-5}	0.041	2.30E-06
	QSS2	10^{-2}	0.242	3.02E-03
	QSS2	10^{-3}	0.891	3.04E-04
	QSS2	10^{-4}	3.063	3.00E-05
OpenModelica	**DASSL**	10^{-3}	0.265	3.80E-03
	DASSL	10^{-4}	0.281	5.40E-04

Rectificador de Media Onda

El rectificador de media onda posee sólo una variable de estado que es el voltaje en el capacitor C1. El modelo fue simulado durante 1 segundo con una tolerancia de 10^{-4}.

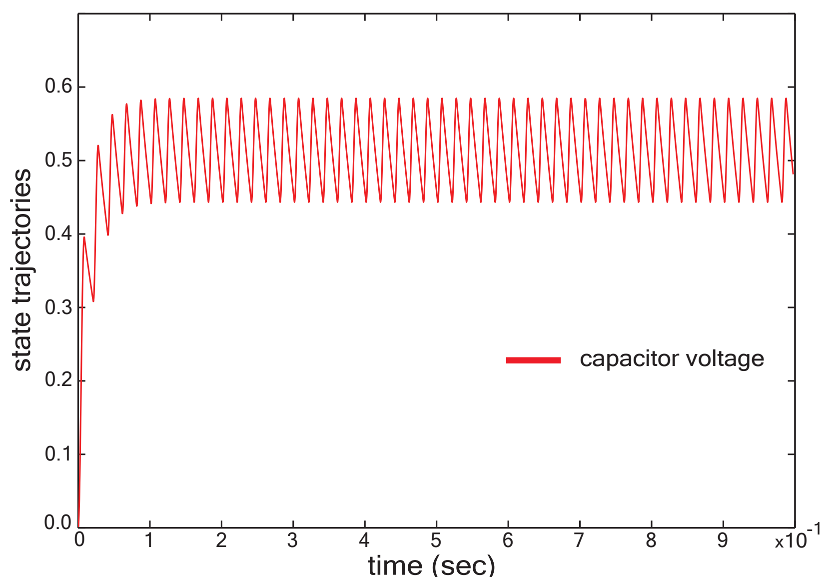

Figura 6.4: Resultado de la simulación utilizando QSS3 para el rectificador de media onda.

En la Figura 6.4 vemos el resultado de la simulación mientras que en la Tabla 6.1 comparamos los distintos algoritmos con las siguientes conclusiones.

Hay una **gran diferencia en la eficiencia de ejecución entre Dymola y OpenModelica** utilizando el método DASSL, siendo Dymola cerca de 10 veces más rápido que OpenModelica a pesar que ambas herramientas usan el mismo solver y el mismo *root solver* (para detectar las discontinuidades). Suponemos que la diferencia se debe al hecho de que OMC no realiza **tearing** [21]. Por ello la resolución de lazos algebraicos se vuelve mucho más costosa y la integración en sí toma más tiempo ya que el número de variables de iteración en DASSL es igual al número de variables de estado más el número de variables de *tearing*. Sin tearing, DASSL debe incluir todas las variables que aparecen involucradas en el lazo algebraico en el conjunto sobre el cual se itera.

Por otro lado, aunque las simulaciones realizadas con QSS3 están basadas en el código generado por OMC, vemos que QSS3 es más eficiente que DASSL en Dymola. Para alcanzar una solución con error en el orden de 10^{-4}, QSS3 requiere 0.014 segundos mientras que DASSL 0.022 segundos. Por lo tanto, **el uso de la interfaz OMPD y la simulación en PowerDEVS utilizando QSS3 acelera la simulación por un factor de 20** comparado con OpenModelica. Notemos que no es justo comparar QSS3 con la versión DASSL de Dymola por lo discutido antes, esto es, la falta de tearing en OMC. La solución de los lazos algebraicos en QSS3 está basada también en el código generado por OMC, por lo tanto la ineficiencia en la generación de código de OMC están siendo propagadas a la simulación con QSS3.

Por esta razón, **debemos comprar los resultados en PowerDEVS con los obtenidos en OpenModelica y no con los de Dymola**. De todas formas es

alentador ver que la ganancia de eficiencia obtenida sobre la simulación estándar de OMC al utilizar métodos de QSS es más eficiente incluso que la simulación obtenida con la herramienta comercial Dymola. Si los métodos de QSS estuvieran implementados en Dymola, los resultados obtenidos por ellos serían de nuevo más eficientes que el obtenido por Dymola actualmente.

Realizando una comparación interna entre los métodos de QSS, es evidente que QSS3 es mucho más eficiente que QSS2. Esto es lógico ya que QSS2 necesita realizar pasos más pequeños comparado con QSS3 para obtener la precisión deseada. **Por ello, podemos concluir que el método de tercer orden QSS3 debería ser el preferido para aplicaciones prácticas**.

Para este ejemplo en particular, RadauIIa no provee una solución correcta a menos que la tolerancia sea del orden de 10^{-7}. Si se utiliza una tolerancia más grande el método trata de utilizar pasos grandes y falla al detectar muchas de las discontinuidades. La detección de discontinuidades utilizada por Dymola no es lo suficientemente robusta. Este problema empeoró considerablemente entre Dymola 6 y Dymola 7, esto es, mientras que RadauIIa fallaba al detectar algunas discontinuidades en Dymola 6, la versión de Dymola 7 falla en detectar muchas más. Esto es un problema serio que la compañía Dinasim debería corregir. Observamos el mismo problema para Dopri45, cuando la tolerancia fue 10^{-3}. Debido a estos problemas, ambos métodos de Runge-Kutta requieren tiempos de CPU comparable al que requiere el método DASSL. Las ventajas inherentes de utilizar métodos monopaso (RadauIIa y Dopri45) contra un método multipaso (DASSL) en un sistemas con discontinuidades frecuentes no pueden explotarse debido a que la implementación actual no puede detectar los eventos correctamente.

Tabla 6.2: Esta tabla muestra los resultados de simulación para distintos algoritmos de simulación para el ejemplo de la fuente conmutada para un tiempo de simulación de 0.01 segundo. Las comparaciones muestran el tiempo de CPU (en segundos) junto con la precisión obtenida comparada contra la trayectoria de referencia con Dymola.

			Tiempo CPU (seg)	**Error de Simulación**
Dymola	**DASSL**	10^{-3}	0.051	1.82E-04
	DASSL	10^{-4}	0.063	7.18E-05
	Radau IIa	10^{-3}	0.064	1.11E-07
	Radau IIa	10^{-4}	0.062	1.11E-07
	Dopri45	10^{-3}	0.049	6.38E-06
	Dopri45	10^{-4}	0.047	9.76E-06
PowerDEVS	**QSS3**	10^{-3}	0.049	1.41E-03
	QSS3	10^{-4}	0.062	1.68E-05
	QSS3	10^{-5}	0.250	8.96E-06
OpenModelica	**DASSL**	10^{-3}	50.496	-
	DASSL	10^{-4}	1.035	?

Fuente Conmutada

El modelo de la fuente conmutada tiene dos variables de estado, la corriente a través del inductor L1 y el voltaje en el capacitor C1. De la Figura 6.3 vemos que

hay una fuente de onda cuadrada (sacada de la Librería Estándar de Modelica) que hace uso de la sentencia `sample` de Modelica. Como esta sentencia no está implementada todavía en la interfaz, la reemplazamos por un sistema marginalmente estable de segundo orden descripto en Modelica como:

```
model PowerConverter
  Real x1(start=0.0);
  Real x2(start=1.0);
  Boolean pulse(start=true);
  parameter Real freq=1e4;
equation
  der(x1)=freq*4*x2;
  der(x2)=if (x1<0) then freq*4 else -freq*4;
  pulse=(x1>0);
  idealClosingSwitch.control = pulse;
end PowerConverter;
```

Notemos que este reemplazo agrega dos nuevas variables de estado $(x1, x2)$ e incrementa el tiempo de simulación ya que el método debe simular también el sistema marginalmente estable. La solución elegida no es única. La conmutación podría haber sido codificada de muchas formas.

El sistema fue simulado por 0.01 segundos y la Figura 6.5 muestra la trayectoria de las variables de estado usando el método QSS3 con una tolerancia de 10^{-4}. La comparación de los resultados obtenidos con distintos métodos de integración se muestran en la Tabla 6.2.

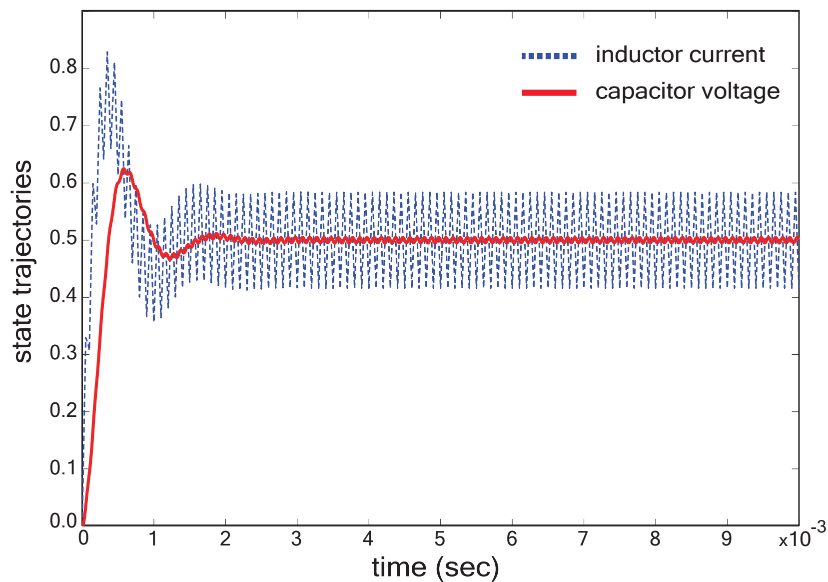

Figura 6.5: Trayectorias de los estados utilizando QSS3 para el modelo de la fuente conmutada.

Las conclusiones obtenidas en el análisis del ejemplo del rectificador de media onda también son válidas en este caso como lo muestra la Tabla 6.2. **El método**

de QSS3 es tan eficiente como el método DASSL de Dymola, mientras que supera a DASSL de OpenModelica y al método de segundo orden QSS2. RadauIIa y Dopri45 simulan correctamente este modelo incluso con valores de tolerancia grandes pero su tiempo de ejecución no mejora significativamente con respecto al de DASSL o QSS3.

Para la fuente conmutada, el error de simulación estimado para DASSL de OpenModelica es bastante grande. Esto es sospechoso ya que no debería ser así. Notamos que en OpenModelica para una tolerancia no tan chica de 10^{-3} la simulación requiere 50 segundos de CPU. Los archivos de salida generado son gigantes, con un tamaño cercano a los 500MB, lo que hace imposible verificar si las trayectorias son correctas o no. Parece haber algo mal en el código generado por OMC para este caso pero no podemos asegurar nada acerca de esto todavía.

6.2. Simulación Autónoma de Modelos Modelica

Hay muchas razones para desear la simulación eficiente de sistemas dinámicos híbridos. Hoy en día la atención está enfocada en varios aspectos de la paralelización del proceso de simulación sin tocar el núcleo de la simulación, es decir el método de integración (o solver). De hecho para muchos investigadores el problema de definir un método de integración eficiente de propósito general para DAE está ya resuelto siendo DASSL el método por defecto para la mayoría de las herramientas comerciales de simulación. Aparte de DASSL, existe una gran variedad de métodos numéricos apuntados a distintas requisitos y familias de modelos.

Postulamos que la atención debería ser puesta de nuevo en lo "básico" y cuestionar la utilización de la **discretización temporal** que utilizan los métodos clásicos. Ya a fines de los 90's, Zeigler introdujo una nueva clase de algoritmos de integración numérica basados en la **cuantificación de estado** y en el formalismo DEVS [95]. Mejorando la idea original de Zeigler, Kofman desarrolló una familia de métodos de QSS que incluyen métodos explícitos para sistemas stiff y no stiff, sistemas marginalmente estables, etc [22, 52, 54, 55, 64]. Ya hemos mencionado las ventajas que esta familia de métodos de QSS posee sobre los métodos clásicos [29, 53, 54, 64].

Originalmente, los métodos de QSS fueron implementados en motores de simulación DEVS como PowerDEVS [11]. Mientras que estas implementaciones eran correctas, algunas propiedades de los motores DEVS introducen grandes retardos (overheads) relacionados a la comunicación de eventos entre distintos niveles del modelo.

Recientemente se desarrolló una familia de métodos de QSS autónomos (i.e. independientes de PowerDEVS) con el fin de solucionar este problema [28]. Estos nuevos métodos alcanzan una mejora en performance de un orden de magnitud en comparación con su implementación DEVS análoga. El simulador autónomo de QSS simula modelos descriptos en una interfaz C que contiene el sistema en forma de ODE, las funciones de cruce por cero junto con información adicional estructural necesaria por los métodos de QSS. Esta interfaz C puede ser generada automáticamente a partir de una descripción de la ODE con una herramienta desarrollada con ese fin.

Ha habido intentos previos de simular modelos Modelica con los métodos de QSS [10, 30] . En la sección previa presentamos una interfaz entre OpenModelica y PowerDEVS (**OMPD interface**) tomando el primer paso a utilizar los métodos de

QSS en la simulación de modelos Modelica. Esta interfaz permite la transformación automática de modelos Modelica al formalismo DEVS para luego ser simulados en PowerDEVS. La interfaz es funcional pero tiene la desventaja del costo introducido por la simulación DEVS antes mencionado.

En esta sección presentamos una extensión al compilador de OpenModelica para traducir modelos Modelica a un **subconjunto del lenguaje Modelica** que llamamos μ–Modelica. Luego desarrollamos una herramienta que a partir de esta descripción del modelo en μ–Modelica genera automáticamente la interfaz C esperada por el simulador autónomo de QSS y luego es simulada por el simulador autónomo

De este modo, este trabajo permite al usuario de Modelica explotar los beneficios de los métodos de QSS directamente desde el entorno de OpenModelica sin ningún conocimiento de DEVS o QSS, utilizándolos como cualquier otro método de integración.

Realizamos también un estudio extenso comparando performance entre los métodos de QSS y el método DASSL utilizado por OpenModelica con dos modelos discontinuos. Los resultados muestran una mejora substancial tanto en términos de eficiencia como de robustez.

6.2.1. Simulador Autónomo de Métodos de QSS

El simulador autónomo de QSS [28] es una herramienta que implementa la familia completa de los métodos de QSS sin utilizar un motor de simulación DEVS.

La herramienta está compuesta por dos módulos:

1. El motor de simulación que integra la ecuación $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{q}, t)$ suponiendo que la trayectoria cuantificada $\mathbf{q(t)}$ está dada.

2. Los solvers que dada la función $\mathbf{x}(t)$ calcula efectivamente $\mathbf{q(t)}$ usando el método de QSS correspondiente.

Una característica importante de los métodos de QSS es que las variables de estado son actualizadas en distintos instantes de tiempo. En cada paso de simulación sólo algunas componentes de $\mathbf{f}(\mathbf{q}, t)$ son evaluadas, por lo tanto el motor de simulación requiere que el modelo esté descripto de forma que pueda evaluar cada componente de $\mathbf{f}(\mathbf{q}, t)$ por separado. Cada función de cruce por cero debe ser también provista como funciones separadas junto con su respectivo manejador (o "handler"). Adicionalmente, se debe proveer información estructural describiendo las dependencias entre variables y ecuaciones.

Todo el entorno de simulación, incluyendo el motor de simulación, los solvers y los modelos están escritos en C.

Como sería bastante incómodo para el usuario describir los modelos proveyendo toda esta información, el simulador autónomo incluye una herramienta que produce esta interfaz C con toda la información estructural desde una descripción de una ODE.

Esta descripción de ODE puede tener los siguientes componentes:

- Ecuaciones ODE de la forma $\dot{x}_j = f_j(\mathbf{x}, \mathbf{a}, \mathbf{d}, t)$ donde $\mathbf{x}$ son estados continuos, $\mathbf{a}$ son variables algebraicas y $\mathbf{d}$ son variables de estado discretas.

- Ecuaciones algebraicas de la forma $a_j = g_j(\mathbf{x}, \mathbf{a}, \mathbf{d}, t)$ con la restricción que a_j sólo puede depender de $a_1, \cdots, a_{j-1}$.

- Funciones de cruce por cero de la forma $z_j = h_j(\mathbf{x}, \mathbf{a}, \mathbf{d}, t)$.
- Asociada a cada función de cruce por cero, dos manejadores o "handlers" (uno para cruce positivo y otro para cruce negativo) donde las variables discretas y continuas pueden ser actualizadas.

Esta descripción es procesada luego por una herramienta que calcula la estructura incluyendo:

- Matrices de incidencia de variables continuas y discretas a ecuaciones ODE,
- Matrices de incidencia de variables continuas y discretas a funciones de cruce por cero,
- Matrices de incidencia de los handlers a las ecuaciones ODE y funciones de cruce por cero.

Esta información es luego utilizada por un generador de código que produce la interfaz C que describe el modelo.

6.2.2. Simulación de Modelos Modelica con el Simulador Autónomo de QSS

Como dijimos antes, desarrollamos primero un lenguaje llamado μ–Modelica y luego extendimos esta herramienta del simulador autónomo para comprender este lenguaje y convertirlo en la descripción ODE.

Extendimos también el OMC para generar modelos en μ–Modelica a partir de modelos Modelica comunes.

De esta forma, un modelo Modelica común puede ser traducido automáticamente y luego simulado por el simulador autónomo de QSS.

En la Figura 6.6 vemos todas las etapas de compilación y simulación involucradas.

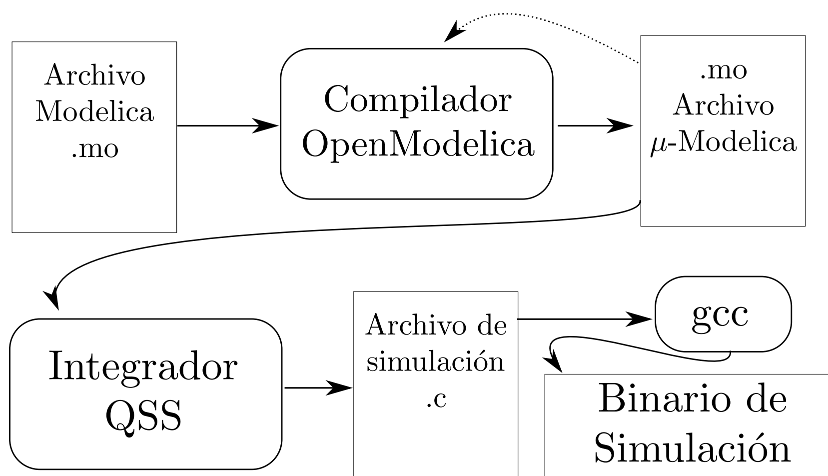

Figura 6.6: Etapas del proceso de compilación y simulación.

En las siguientes secciones introducimos primero el lenguaje μ–Modelica y luego describimos el proceso de traducción de Modelica a μ–Modelica.

6.2.3. El Subconjunto μ–Modelica

El lenguaje μ–Modelica fue definido como un subconjunto de Modelica lo más cercano posible a la descripción ODE aceptada por el simulador autónomo de QSS. μ–Modelica contiene sólo las palabras claves y estructuras necesarias para definir un modelo híbrido en forma de ODE. La expresividad de μ-Modelica es menor que la de Modelica ya que el primero no permite el modelado orientado a objetos ni descripción de ecuaciones diferenciales algebraicas.

El lenguaje μ-Modelica tiene las siguientes restricciones:

- El modelo está ya aplanado (ver Sección 6.1.2), esto es, no hay información de clases.
- Todas las variables son del tipo `Real` y hay sólo tres clases de variables: estados continuos (`x[]`), estados discretos (`d[]`) y algebraicas (`a[]`).
- Los parámetros son también de tipo `Real` y pueden tener cualquier nombre.
- Las ecuaciones están dadas en forma de ODE **explícitas**.
- Una variable algebraica `a[i]` sólo puede depender de variables algebraicas definidas previamente (`a[1:i-1]`).
- Las discontinuidades son expresadas sólo por cláusulas `when` dentro de la sección `algorithm`. Las condiciones dentro de las cláusulas `when` sólo pueden ser comparaciones ($<, \leq, >, \geq$) y dentro de las cláusulas sólo pueden asignarse variables discretas (`d[]`) y continuas a través de `reinits`.

Este lenguaje restringido no está pensado para ser usado por un usuario, sino sólo como lenguaje intermedio entre OpenModelica y el simulador autónomo de QSS. El usuario normal debería utilizar el lenguaje Modelica completo y luego usar el OMC para obtener una versión en μ-Modelica.

6.2.4. Simulando Modelos μ–Modelica con el Simulador Autónomo de QSS

Como mencionamos antes, el simulador autónomo de QSS incluye un analizador sintáctico de código que extrae toda la información estructural necesaria de una representación ODE. Este analizador sintáctico fue extendido para comprender el lenguaje μ–Modelica. Esta herramienta realiza las siguientes acciones:

- Reconoce las palabras claves de Modelica para `parameter`, y estados discretos (`discrete`).
- Analiza las ecuaciones de la forma `der(x[i])=expr()`, generando la ODE correspondiente junto con la información estructural.
- Reconoce las cláusulas de la forma `when expr1>expr2 then`, generando las funciones de cruce por cero `zc=expr1-expr2` con un manejador para el cruce positivo en el cual incluirá todas las sentencias que estén dentro de la cláusula. Si encuentra una cláusula `elsewhen expr1<expr2 then`, genera también el manejador para el cruce negativo.
- Genera también la información correspondiente a las funciones de cruce por cero y sus manejadores.

6.2.5. Conversión de Modelica a μ–Modelica

Para completar el proceso de simulación de un modelo Modelica normal con el simulador autónomo de QSS agregamos un nuevo *target o tipo de salida* al compilador de OpenModelica para que genere la salida en el lenguaje μ-Modelica.

Mucho del trabajo es realizado ya por OMC sin modificaciones. Primero simplifica expresiones, ordena las ecuaciones y transforma la DAE en una ODE produciendo el código necesario para resolver los lazos algebraicos. También reconoce las discontinuidades y genera las funciones de cruce por cero correspondientes.

Tomamos entonces la estructura generada por OMC y la procesamos de la siguiente forma:

1. Identifica las variables de estado continuas (aquellas que son utilizadas en el operador `der`), variables algebraicas (aquellas que se resuelven en ecuaciones ODE pero no son estados) y las variables de estado discretas (aquellas definidas como discretas, incluyendo variables de tipo `Integer` y `Boolean`). Las variables booleanas son remplazadas por variables reales donde `1.0` representa verdadero y `0.0` falso.

2. Los nombres de los parámetros son modificados reemplazando el/los punto(s) por guión bajo. Esto se realiza para todos los identificadores.

3. En cada ecuación de la parte ODE, se reemplaza cada aparición de una variable de estado continua, discreta y algebraicas por su alias de μ–Modelica correspondiente `x[]`, `a[]` o `d[]`.

4. Si la ecuación es parte de un lazo algebraico, se genera una función externa en C que resuelve el lazo y se emite una llamada a esta función en el archivo μ-Modelica.

5. Para cada función de cruce por cero se generan cláusulas `when` y `elsewhen`. El `elsewhen` extra es necesario para asignar diferentes valores a las variables discretas asociadas con la función de cruce.

6. Cláusulas `when` son emitidas reemplazando las variables de estado continuas, discretas y algebraicas por sus alias en la condición de la cláusula y también en el cuerpo de ésta.

7. Los operadores `sample` son eliminados y reemplazados utilizando una variable de estado discreta extra.

8. Las cláusulas `elsewhen` son emitidas como clausulas `when` normales en la sección `algorithm`.

Por ejemplo el modelo de una pelota rebotando en Modelica:

```
model bball1
  Real y(start = 1),v,a;
  Boolean flying(start = true);
  parameter Real m = 1;
  parameter Real g = 9.8;
  parameter Real k = 10000;
  parameter Real b = 10;
equation
```

```
  der(y) = v;
  der(v) = a;
  flying = y>0;
  a = if flying then -g else -g -
       - (b * v + k * y)/m;
end bball1;
```

se traduciría a μ-Modelica como:

```
model bball1
  constant Integer N = 2;
  Real x[N](start=xinit());
  discrete Real d[1](start=dinit());
  Real a[1];
  parameter Real m = 1.0;
  parameter Real g = 9.8;
  parameter Real k = 10000.0;
  parameter Real b = 10.0;
  function xinit
    output Real x[N];
  algorithm
    x[2]:= 1.0 /* y */;
    x[1]:= 0.0 /* v */;
  end xinit;
  function dinit
    output Real d[1];
  algorithm
    d[1]:=(1.0) /* flying*/;
  end dinit;
/* Equations */
equation
  der(x[2]) = x[1];
  a[1] = -d[1] * g + (1.0 - d[1]) *
  (((-b) * x[1] + (-k) * x[2]) / m - g);
  der(x[1]) = a[1];
algorithm
/* Discontinuities */
  when x[2] > 0.0 then
    d[1] := 1.0;
  elsewhen x[2] < 0.0 then
    d[1] := 0.0;
  end when;
end bball1;
```

Vemos fácilmente que el modelo contiene dos variables de estado continuas, una algebraica y una variable de estado discreta junto con una discontinuidad sobre `x[2]` que actualiza la variable de estado discreta.

Cuando el modelo Modelica contiene un lazo algebraico, éste será detectado por OMC y μ-Modelica incluirá código de la forma:

```
...
  function fsolve15
    input Real i0;
    input Real i1;
```

```
    output Real o0;
    output Real o1;
    output Real o2;
  external "C" ;
  end fsolve15;
...
equation
...
(a[1],a[2],a[3])=fsolve15(x[2],d[1])
```

junto con una función C que resuelve el lazo utilizando la librería GNU Scientific Library (GSL) [36].

Esta llamada indica que las variables `a[1:3]` son calculadas juntas por una función externa C, por lo cual el analizador de código del simulador autónomo de QSS lo trata como una llamada a una función normal al computar la información estructural.

En la función externa antes mencionada, mejoramos lo que hace OMC teniendo en cuenta una característica de los lazo algebraicos lineales. Un lazo algebraico lineal generalmente tiene la forma $A \cdot z = b$ (z es la incógnita), donde A depende solamente de variables discretas. Luego, cuando el cambio en la variable de estado continuo sólo afecta el término b, **no es necesario** invertir la matriz en ese paso de integración, ahorrando mucho cálculo.

6.2.6. Ejemplos y Resultados

En esta Sección analizamos los resultados obtenidos utilizando la herramienta presentada. Como casos de estudio analizamos dos sistemas que exhiben discontinuidades frecuentes, una fuente conmutada tipo buck y un circuito DC-DC interleaved. Ambos modelos fueron desarrollados utilizando la Modelica Standard Library 3.1 y pueden ser descargados de [8]. Para cada ejemplo usamos la versión modificada de OMC (r11645) para generar la versión μ-Modelica correspondiente y luego simulados con el simulador autónomo de QSS. En cada caso, comparamos la eficiencia de run-time y precisión de los métodos de QSS contra el método DASSL de OpenModelica v1.8.1. Para medir el tiempo de ejecución de cada método, utilizamos el tiempo reportado por cada entorno. Aunque OpenModelica provee varias formas de medir el tiempo de CPU necesario para la simulación (incluyendo un "profiler" [1]) observamos diferencias importantes entre los tiempos reportados. Luego de consultarlo con los desarrolladores de OpenModelica, medimos el tiempo ejecutando la simulación con `time ./model_executable -lv LOG_STATS` para medir sólo el tiempo de simulación. Notamos aquí que el tiempo reportado de este modo es mucho menor que el reportado por el intérprete interactivo de OpenModelica, OMShell y que el reportado por el profiler. Luego, los resultados de aceleración que obtenemos pueden ser considerados conservadores.

En estos casos la plataforma de ejecución es la misma que en la Sección 6.1.3. De nuevo, el tiempo de CPU no debe ser considerado como una magnitud absoluta ya que variará de una computadora a otra pero el orden relativo de los algoritmos se mantendrá.

De nuevo la precisión de los algoritmos sólo pode ser calculada aproximadamente ya que las trayectorias no pueden ser no puede ser calculadas analíticamente.

[1] Un profiler es una herramienta de análisis de rendimiento

Utilizamos aquí trayectorias de referencias ($\mathbf{t^{ref}}, \mathbf{y^{ref}}$) calculadas con LIQSS2 y una tolerancia de 10^{-7}. El error de simulación es también computado como lo hicimos en la Sección 6.1.3.

Fuente Conmutada

En este caso estudiaremos el ejemplo visto en la Sección 6.1.3 sólo que en este caso cambiamos los parámetros R y C de manera que ahora $R = 10, C = 0{,}0001$.

Con estos nuevos parámetros el sistema se vuelve stiff debido a que el circuito entra en modo de conducción discontinua, es decir, hay instantes de tiempo en los cuales la llave y el diodo de la Figura 6.3 se encuentran abiertos en forma simultánea.

El modelo fue simulado 0.01 segundos utilizando distintos métodos aptos para sistemas stiff. La trayectoria de referencia puede verse en la Figura 6.7.

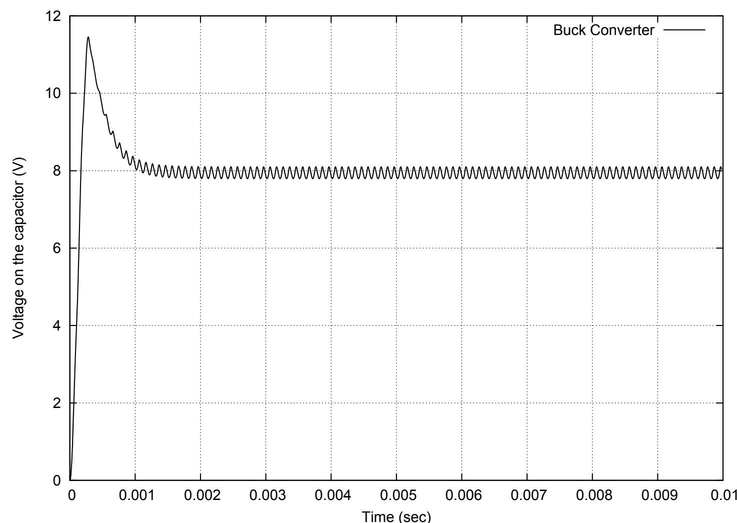

Figura 6.7: Circuito Fuente Conmutada Buck - Simulación.

Inicialmente simulamos el modelo en OpenModelica utilizando el número de puntos de salida por defecto (500). Observamos que el método DASSL de OpenModelica no detecta correctamente los eventos. Por otro lado, cuando forzamos OMC a generar más puntos de salida, el error decrementa ya que las evaluaciones extras necesarias para generar los puntos de salida **fuerza** DASSL a re-evaluar las funciones de cruce por cero detectando ahí los eventos. Por esta razón comparamos la versión DASSL de OpenModelica con distintos números de puntos de salida y distinta precisión contra los métodos de QSS utilizando los métodos aptos para stiff LIQSS2 y LIQSS3. Los resultados están resumidos en la Tabla 6.3.

Observamos que para 500 puntos de salida DASSL no logra reducir el error de simulación al achicar la precisión. Esto es un claro signo de que la simulación no es correcta. Cuando la cantidad de puntos de salida es incrementada a 10000

los resultados de OMC se acercan a las trayectorias de referencias y el error de simulación se reduce.

Tabla 6.3: Esta tabla muestra los resultados de simulación para varios métodos para el circuito de la Fuente Conmutada Buck para un tiempo de simulación de 0.01 segundos. La comparación realizada incluye el tiempo de CPU requerido (en msec), el número de pasos de integración junto con la precisión relativa de la simulación comparada con las trayectorias de referencia obtenidas con LIQSS2 y una tolerancia de 10^{-7}.

			500 output points			**10000 output points**		
			Tiempo de CPU (mseg)	**Pasos**	**Error de Simulación**	**Tiempo de CPU** (mseg)	**Pasos**	**Error de Simulación**
QSS	**LIQSS3**	10^{-2}	4	3351	5.84E-03	4	3351	5.83E-03
	LIQSS3	10^{-3}	8	4163	7.31E-04	8	4163	7.32E-04
	LIQSS3	10^{-4}	12	6804	4.60E-05	12	6804	4.61E-05
	LIQSS3	10^{-5}	20	11314	1.07E-06	20	11314	1.08E-06
	LIQSS2	10^{-2}	4	3863	7.83E-03	4	3863	7.84E-03
	LIQSS2	10^{-3}	8	6715	1.32E-03	8	6715	1.32E-03
	LIQSS2	10^{-4}	12	18519	1.15E-04	12	18519	1.15E-04
	LIQSS2	10^{-5}	32	53391	6.42E-06	32	53391	6.42E-06
OpenModelica	**DASSL**	10^{-3}	22	4273	3.56E-03	70	5249	2.66E-04
	DASSL	10^{-4}	28	5636	3.17E-03	72	5955	1.75E-04
	DASSL	10^{-5}	32	7781	3.28E-03	74	7623	2.40E-05

Por ello, tiene sentido comprar el tiempo de ejecución para casos de 10000 puntos donde vemos claramente que los métodos de QSS son más eficientes que DASSL. Para realizar una simulación para un error en el orden de 10^{-5}, LIQSS3 requiere 12 mseg mientras que DASSL necesita 74 mseg. Por lo tanto **el uso de LIQSS3 en vez del método estándar DASSL acelera la simulación por un factor de 6x**. La reducción de tiempo de ejecución y precisión se muestra en al Figura 6.8. Los resultados están graficados con escalas logarítmicas donde cuanto más cerca las líneas están del origen mejor se desempeña el algoritmo.

Realizando una comparación interna entre ambos métodos de QSS, vemos que el método de tercer orden LIQSS3 es levemente más eficiente que LIQSS2, especialmente cuando los requisitos de precisión, es decir el error realizado, se achica. Esto es de esperar ya que LIQSS2 necesita realizar pasos más chicos en comparación a LIQSS3 para obtener la precisión deseada (por ejemplo para un error de 10^{-6} LIQSS2 realiza 53391 pasos mientras que LIQSS3 sólo hace 11314). **Podemos concluir que el método de tercer orden LIQSS3 debería ser el preferido para aplicaciones prácticas**. Vemos también que como los algoritmos de QSS proveen salida densa, el número de puntos de salida no afecta los tiempos de simulación.

Finalmente, otra característica de los métodos de QSS es evidente de los resultados obtenidos. Verificamos que en general DASSL realiza menos pasos que cualquier de los métodos de QSS aunque los pasos de DASSL son mucho más costosos y complicados que los realizados por QSS ya que los primeros requieren -en general- la estimación de la función completa $\mathbf{f}(\cdot)$.

Por otro lado, en cada paso de QSS sólo se actualiza una variable de estado, lo cual requiero sólo re-evaluar la componente correspondiente de $\mathbf{f_i}(\cdot)$. En cuanto los sistemas simulados son más grandes, más complejos y más ralos evaluar $\mathbf{f_i}(\cdot)$ es mucho más eficiente que evaluar la $\mathbf{f}(\cdot)$ completa.

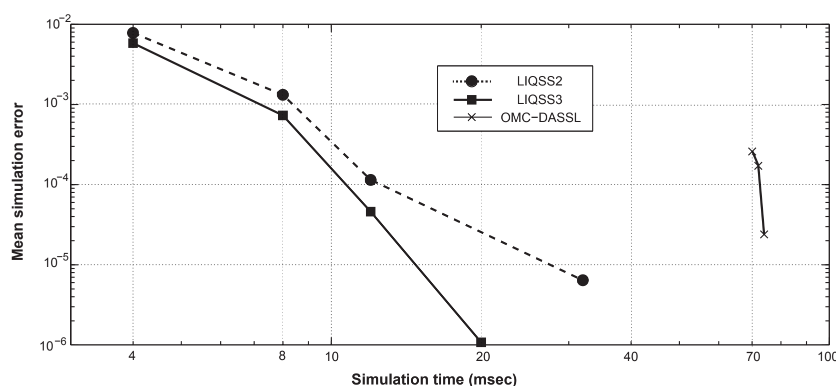

Figura 6.8: Tiempo de CPU vs Error para el ejemplo Fuente Conmutada Buck (10000 puntos de salida).

Circuito Interleaved DC-DC

La Figura 6.9 muestra el modelo de un circuito convertidor “buck interleaved”. El circuito es similar a la Fuente Conmutada Buck analizando previamente pero contiene varias secciones conmutadas que son activadas en distintos momentos para

reducir el ripple en el voltaje de salida. En este caso, consideramos un circuito con cuatro ramas.

Para construir este modelo, todos los componentes fueron tomados de la Modelica Standar Library 3.1 excepto por el `booleanDelay` que implementa un retardo booleano que emite su entrada luego de un período fijo `T`. Este retardo no tiene memoria, esto es, cuando se recibe una entrada, cualquier salida agendada es cancelada y sobreescrita por la nueva entrada.

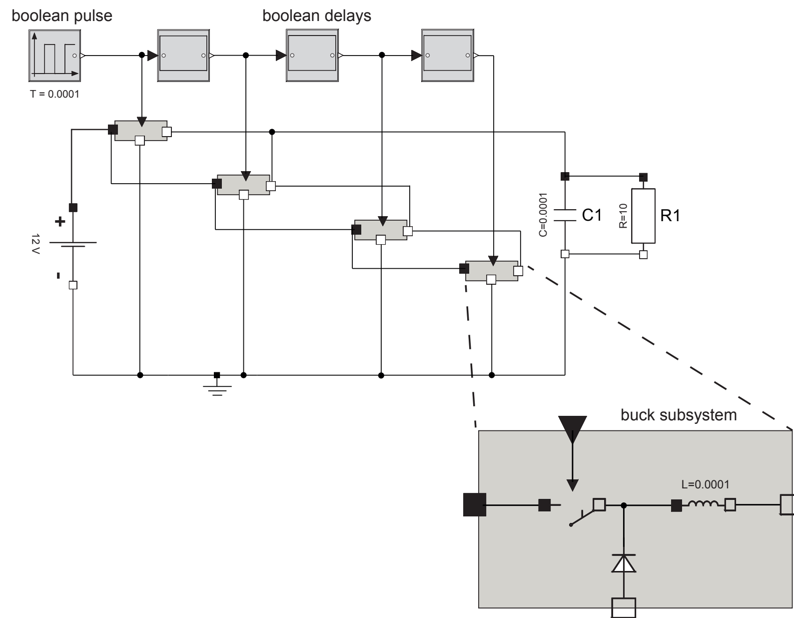

Figura 6.9: Circuito DC-DC interleaved.

Simulamos este modelo durante 0.01 seg. nuevamente fijando nuestra atención en el voltaje en el capacitor obteniendo la trayectoria de simulación de la Figura 6.10. Realizamos los mismos experimentos que en el ejemplo anterior (Sección 6.2.6) y los resultados obtenidos se muestran en la Tabla 6.4.

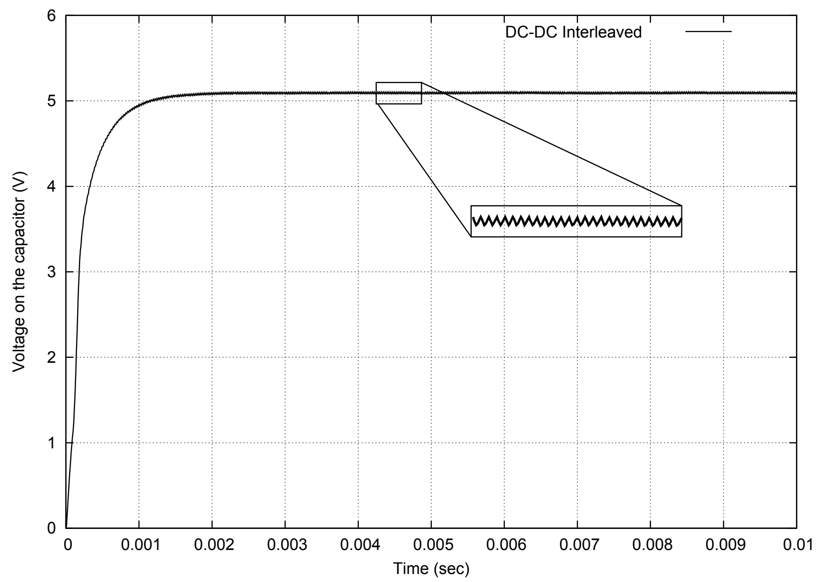

Figura 6.10: DC-DC Interleaved - Simulación.

Tabla 6.4: Esta tabla muestra los resultados de simulación para varios métodos para el ejemplo del circuito DC-DC interleaved para un tiempo de simulación de 0.01 seg. Las comparaciones incluye el tiempo de CPU (en mseg), la cantidad de paso realizados, y la precisión de la simulación en comparación con trayectorias de referencias obtenidas con LIQSS2 con una tolerancia de 10^{-7}.

			500 puntos de salida			**10000 puntos de salida**		
			Tiempo de CPU (mseg)	**Pasos**	**Error de Simulación**	**Tiempo de CPU** (mseg)	**Pasos**	**Error de Simulación**
QSS	**LIQSS3**	10^{-2}	32	18396	1.32E-02	32	18396	1.32E-02
	LIQSS3	10^{-3}	60	33426	7.31E-04	60	33426	7.31E-04
	LIQSS3	10^{-4}	48	29408	1.57E-04	48	29408	1.57E-04
	LIQSS3	10^{-5}	64	39951	6.48E-06	64	39951	6.48E-06
	LIQSS2	10^{-2}	12	10715	4.08E-03	12	10715	4.08E-03
	LIQSS2	10^{-3}	20	29082	3.63E-04	20	29082	3.63E-04
	LIQSS2	10^{-4}	56	73218	1.26E-04	56	73218	1.26E-04
	LIQSS2	10^{-5}	128	198001	8.80E-06	128	198001	8.80E-06
OpenModelica	**DASSL**	10^{-3}	310	14421	4.96E-02	428	17571	2.37E-02
	DASSL	10^{-4}	363	22375	5.03E-02	442	18574	2.37E-02
	DASSL	10^{-5}	496	31387	5.41E-02	488	23625	5.57E-03

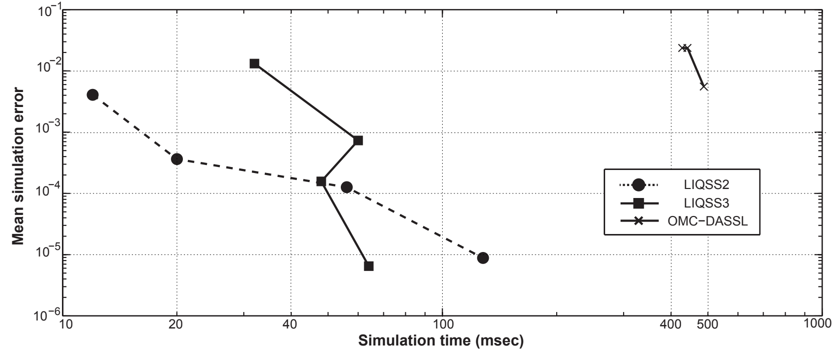

Figura 6.11: Tiempo de CPU vs Error para el modelo DC-DC interleaved (10000 puntos de salida).

Vemos en la Figura 6.11 que para obtener un error de simulación del orden de 10^{-3} el método DASSL requiere 488 mseg mientras que LIQSS2 requiere 12 mseg y LIQSS3 60 mseg. **Esto muestra una aceleración de 40x y 8x para LIQSS2 y LIQSS3 respectivamente**.

La diferencia de los tiempos entre LIQSS2 y LIQSS3 se debe a que la implementación de LIQSS3 no está completamente optimizada para todos los casos y algunos problemas todavía existen. Aquí también al aumentar la cantidad de puntos de salida en los métodos de QSS ni el error ni la cantidad de pasos varía ya que estos métodos proveen salida densa.

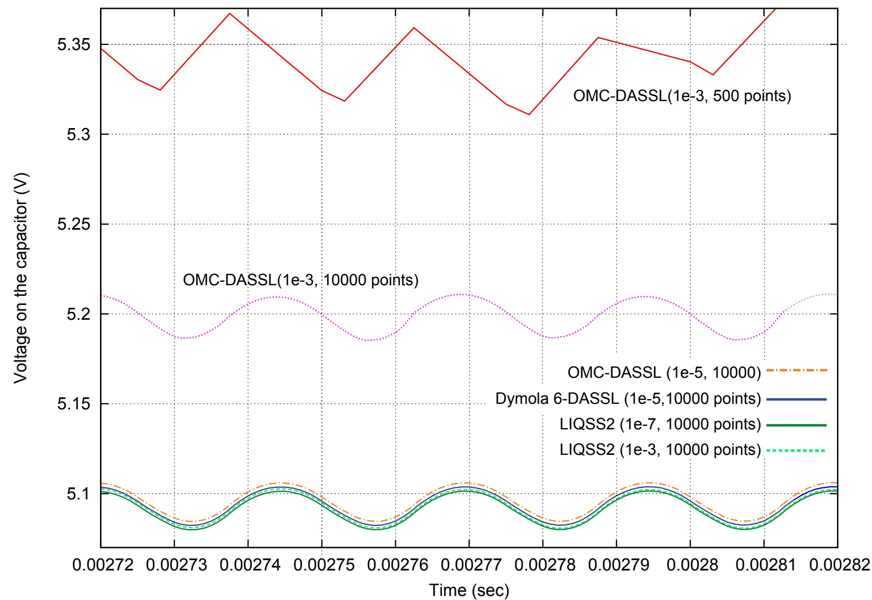

Figura 6.12: Comparación del estado estacionario final para distinto métodos.

En la Figura 6.12 mostramos los distintos régimen de estado estacionario obtenidos con distintos métodos. Vemos nuevamente que la detección de discontinuidades de OpenModelica está fuertemente influenciada por el número de puntos

de salida. Aquí incluimos los resultados de Dymola 6.0 para proveer una solución generalmente aceptada como solución correcta. No hemos realizado experimentos sobre el tiempo de ejecución con Dymola.

6.3. Conclusiones

En este Capítulo presentamos dos metodologías para simular modelos descriptos en el lenguaje Modelica utilizando los métodos de QSS.

En la Sección 6.1 desarrollamos una interfaz entre OpenModelica y PowerDEVS (OMPD interface) que trata correctamente modelos discontinuos y permite simular problemas del mundo real descriptos en Modelica en el software de simulación PowerDEVS.

Realizamos comparaciones en dos casos de estudio demostrando una ganancia en eficiencia al utilizar los métodos de QSS3 sobre el método estándar DASSL. La interfaz propuesta utiliza el código generado por el compilador de OpenModelica por lo cual comparamos la utilización de DASSL y QSS3 con este código. Alcanzamos una reducción de 20 veces en el tiempo de CPU necesario para la simulación.

Además las comparaciones muestran que la eficiencia de QSS3 utilizando el código generado por OpenModelica es comparable al tiempo de simulación de Dymola utilizando DASSL a pesar de que Dymola ofrece un pre procesamiento de modelo mucho más sofisticado como un algoritmo de rasgado para modelos con lazos algebraicos. Por ello pensamos que habría una gran ganancia en la eficiencia de la simulación si la interfaz fuera a ser implementada como parte del compilador de Dymola.

En la Sección 6.2 presentamos y analizamos una segunda metodología para la simulación de modelos Modelica utilizando los métodos de QSS. En este caso no se utiliza PowerDEVS sino que la simulación la realiza un simulador autónomo de QSS. La implementación fue verificada y probamos también su eficiencia simulando modelos Modelica del mundo real.

Las comparaciones realizadas en dos casos de uso demuestran también un aumento de la eficiencia de los métodos LIQSS sobre el método defecto DASSL de OpenModelica.

Se alcanzó una reducción en el tiempo de CPU necesario de hasta 40 veces. Además observamos que para ambos sistemas el método de DASSL no produce el resultado correcto si no se lo fuerza a emitir muchos puntos de salida. Aumentar el número de puntos de salida significa incrementar el número de pasos de integración realizados por DASSL aumentando así el tiempo computacional. Por otro lado, los métodos de QSS no sólo simulan correctamente ambos modelos para todas las configuraciones, sino que al proveer salida densa el número de pasos se mantiene constante sin importar cuántos puntos de salida son generados.

Capítulo 7

Conclusiones Generales

En este Capítulo concluimos la Tesis resumiendo los algoritmos y métodos desarrollados durante este trabajo y los resultados de su aplicación en diversos campos de la ciencia.

Delineamos luego varias líneas de trabajo a futuro y mencionamos algunos problemas abiertos en los cuales pretendemos trabajar.

7.1. Conclusiones Generales

En esta Tesis investigamos diversas técnicas de simulación de sistemas continuos e híbridos mediante eventos discretos y su ejecución en tiempo real y paralelo. Ejecutar una simulación en tiempo real es deseable para realizar experimentos Man In the Loop o Hardware In the Loop en los cuales una persona o un dispositivo de hardware es reemplazado por un prototipo que simula al mismo.

El desarrollo de la plataforma de modelado y simulación en tiempo real PowerDEVS–RTAI presentada en el Capítulo 3 brinda al usuario una herramienta simple y unificada para realizar esta clase de experimentos utilizando la aproximación de los métodos de QSS.

Como los métodos de QSS son explícitos, es decir, no iteran en cada paso, incluso al simular sistemas stiff, el tiempo de cómputo puede acotarse lo cual es un requisito fundamental para llevar a cabo la simulación en tiempo real. La plataforma PowerDEVS–RTAI ofrece también diversos métodos para comunicarse con dispositivos de hardware como escritura a puertos, sincronización temporal, manejo de interrupciones de hardware, etc.

Esta plataforma ha sido utilizada en varios trabajos como base para desarrollar sistemas MIL y HIL [3, 5, 37].

Luego en el Capítulo 4 introdujimos una extensión al formalismo DEVS llamado Vectorial DEVS que apunta al *modelado gráfico* de grandes sistemas continuos e híbridos. Al extender la notación del formalismo DEVS permitimos la descripción de arreglo de modelos que y proveemos también mecanismos para conectarlos entre sí.

Implementamos Vectorial DEVS en PowerDEVS, incluyendo nuevos bloques y desarrollamos varios casos de estudio que muestran este formalismo en acción. La estructura regular de los modelos Vectorial DEVS permiten particionarlos fácil-

mente para luego ser simulados concurrentemente en distintas unidades de procesamiento, es decir en paralelo.

Previo a este trabajo se habían presentado diversas técnicas para la simulación de modelos DEVS en paralelo aunque mucho del trabajo estaba enfocado en sistemas *puramente* discretos. Ninguna de estas técnicas eran aptas para la simulación de sistemas continuos.

En el Capítulo 5 introdujimos SRTS y ASRTS, dos novedosas técnicas para la simulación de sistemas continuos e híbridos en paralelo mediante eventos discretos y métodos de QSS. Implementamos estas técnicas en PowerDEVS utilizando el sistema operativo de tiempo real RTAI y el entorno desarrollado en el Capítulo 3.

Analizamos la aplicación de estas técnicas a tres ejemplos descriptos utilizando modelos Vectorial DEVS y estudiamos su performance y limitaciones obteniendo aceleraciones de entre 4,5 y 9 veces utilizando 12 procesadores físicos. También analizamos el error numérico adicional introducido por estas técnicas. Encontramos que si el factor de escalado de tiempo real r se mantiene lo suficientemente chico para evitar overruns, el error se mantiene acotado. Además ASRTS ajusta r para evitar overruns por lo cual el error introducido por ASRTS es pequeño.

Finalmente en el Capítulo 6 presentamos dos metodologías para simular modelos descriptos en el lenguaje Modelica utilizando los métodos de QSS. La primera consta de una interfaz entre OpenModelica y PowerDEVS (Interfaz OMPD) que simula modelos descriptos en Modelica traduciéndolos a PowerDEVS y luego simulándolos con los métodos de QSS.

La segunda metodología introduce un lenguaje intermedio llamado $\mu{-}Modelica$, el cual es un sub–conjunto de Modelica, utilizado como interface entre OpenModelica y un simulador autónomo de QSS. En este caso no utilizamos código C generado por OpenModelica, el cual tiene problemas de eficiencia, sino que el simulador lo genera a partir de la descripción en $\mu - Modelica$. La implementación fue verificada y probamos también su eficiencia simulando modelos Modelica realistas.

En ambos caso realizamos comparaciones en varios casos de estudio (incluyendo sistemas stiff) demostrando una ganancia en eficiencia al utilizar los métodos de QSS sobre los métodos clásicos (en particular DASSL). En el caso de la interfaz OMPD redujimos el tiempo de simulación hasta 20 veces mientras que utilizando el simulador autónomo la reducción fue de 40 veces.

7.2. Trabajo a Futuro y Problemas Abiertos

En cuanto a trabajo a futuro estamos realizando varios proyectos que continuarán extendiendo los aportes de esta Tesis.

Estamos utilizando el algoritmo de sincronización temporal desarrollado en este trabajo para implementar el control en tiempo real de un motor sobre un procesador de arquitectura ARM con una placa de evaluación que incluye sensores y actuadores. Para ello se debe portar toda la librería de PowerDEVS para ejecutarse dentro de esta plataforma embebida. Este trabajo está siendo realizado en convenio con la empresa Forkworks de Villa Constitución, Santa Fe - Argentina.

El algoritmo de particionado para modelos Vectorial DEVS realiza una división estática, es decir, que se mantiene durante toda la simulación. En algunos casos tiene sentido re–balancear el particionado durante la simulación, por ejemplo, si

algunas partes del sistema se mantienen constantes mientras que otra varían demasiado. Estudiaremos para ello políticas de re–balanceo utilizando las técnicas de paralelismo vistas en el Capítulo 5. También se está realizando trabajo en conjunto con el *Modeling and Simulation Research Group*, ETH Zurich-Suiza, dirigido por François Cellier para desarrollar estas técnicas de re–balanceo y paralelismo a la simulación de modelos Modelica.

En cuanto a las técnicas de paralelización en sí mismas, sería deseable tener un mecanismo de recuperación para las situaciones de overrun. Podríamos acotar el error introducido implementado un mecanismo de recuperación con *rollback* cuando se detectan grandes overruns. Para este fin, podríamos utilizar los checkpoints de Adaptive–SRTS como etapas de rollback. De esta forma, sólo deberíamos salvar el estado del sistema en el último checkpoint, descartándolo en el próximo checkpoint consumiendo una cantidad acotada y constante de memoria. También sabemos exactamente cuanto tiempo desperdiciamos con cada rollback: ΔT.

Trabajaremos también en un análisis formal de cómo afectan los retardos en la comunicación al resultado numérico, siguiendo la idea presentada en la Sección 5.1.3. La idea es presentar los efectos de los retardos como perturbaciones acotada y analizar el sistema perturbado para asegurar su estabilidad numérica y cotas de error.

Un problema abierto es la implementación de estas técnicas en un número mayor de procesadores o en un cluster. Para ello, varios problemas deben ser resueltos primero, como la sincronización de los relojes físicos entre distintos nodos o CPUs, la comunicación entre nodos (en la implementación sobre multicore está implementado utilizando memoria compartida) la asignación de CPU a hilos.

Mientras que hemos desarrollado estas técnicas paralelas para la simulación de sistemas continuos mediante eventos discretos sería interesante estudiar el uso de estas técnicas con otros métodos de integración numérica (en particular con algoritmos multi–rate). También investigar el estudio de las técnicas de SRTS y ASRTS a aplicaciones más generales que la simulación de eventos discretos.

En cuanto a la simulación de modelos Modelica nuestro objetivo más cercano es extender los casos de uso a modelos más complejos. En particular apuntamos a ejecutar todos los modelos de ejemplo de la Modelica Standard Library comprobando en qué casos obtenemos un beneficio y en cuáles no.

Finalmente queremos utilizar las técnicas de paralelización para ejecutar el simulador autónomo de QSS en una arquitectura multicore.

Bibliografía

[1] Martin Adelantado, Pierre Siron, and Jeaunbaptiste Chaudron. Towards an HLA Run-time Infrastructure with Hard Real-time Capabilities. In *International Simulation Multi-Conference*, Ottawa, Canada, 2010.

[2] J. Akesson, M. Gafvert, and H. Tummescheit. JModelica–an Open Source Platform for Optimization of Modelica Models. *Proceedings of MATHMOD 2009 - 6th Vienna International Conference on Mathematical Modelling*, 2009.

[3] J.M. Alvarez Leiva, J. Tarrio, and E. Kofman. Implementación de Equipos de Control no Lineal con Muestreo Asincrónico. In *Proc. of RPIC 2011*, Paraná, Argentina, 2011.

[4] Michael Barabanov. A Linux-based RealTime Operating System. Master's thesis, New Mexico Institute of Mining and Technology, New Mexico, 1997.

[5] Crístian Basabilbaso, Juan Zúccolo, Federico Bergero, and Ernesto Kofman. Simulación en tiempo real de sistemas de control de movimiento. In *RPIC 2009 - XIII Workshop on Information Processing and Control*, Rosario, Argentina, 2009.

[6] D.W. Bauer and E.H. Page. Optimistic parallel discrete event simulation of the event-based transmission line matrix method. In *Winter Simulation Conference*, pages 676–684, 2007.

[7] Federico Bergero. BackDoor en el sitio de Scilab–ATOMS. http://atoms.scilab.org/toolboxes/BackDoor.

[8] Federico Bergero. Modelica models for download at:. http://www.fceia.un.edu.ar/~fbergero/modelica2012.

[9] Federico Bergero. Desarrollo de una Plataforma de Simulación en Tiempo Real por Eventos Discretos. Technical report, FCEIA - UNR, Rosario, Argentina, 2008.

[10] Federico Bergero, Xenofon Floros, Joaquín Fernández, Ernesto Kofman, and François E. Cellier. Simulating Modelica models with a Stand–Alone Quantized State Systems Solver. In *9th International Modelica Conference*, 2012. Aceptado.

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

[12] Federico Bergero, Ernesto Kofman, Crístian Basabilbaso, and Juan Zúccolo. Desarrollo de un simulador de sistemas híbridos en tiempo real. In *Proceedings of AADECA 2008*, Buenos Aires, Argentina, 2008.

[13] Federico Bergero, Ernesto Kofman, and Francois Cellier. A novel parallelization technique for DEVS simulation of continuous and hybrid systems. *Simulation*, 2012.

[14] Federico Bergero, Ernesto Kofman, and Cellier Francois E. Scaled Real–Time Parallelization for DEVS Simulation of Hybrid Systems. In *HPC-LataM*. JAIIO, 2012.

[15] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, 2009.

[16] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, 2009.

[17] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.

[18] Azzedine Boukerche and Carl Tropper. A static partitioning and mapping algorithm for conservative parallel simulations. *SIGSIM Simul. Dig.*, 24(1):164–172, 1994.

[19] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Massachusetts Institute of Technology Cambridge, MA, USA, Cambridge, MA, USA, 1977.

[20] R. Castro, E. Kofman, and F. Cellier. Quantization Based Integration Methods for Delay Differential Equations. *Simulation Modelling Practice and Theory*, 19(1):314–336, 2011.

[21] François Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer, New York, 2006.

[22] François Cellier, Ernesto Kofman, Gustavo Migoni, and Mario Bortolotto. Quantized State System Simulation. In *Proceedings of SummerSim 08 (2008 Summer Simulation Multiconference)*, Edinburgh, Scotland, 2008.

[23] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981.

[24] Alex Chung Hen Chow and Bernard P. Zeigler. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation*, WSC '94, pages 716–722, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[25] E. Deelman and B.K. Szymanski. Dynamic load balancing in parallel discrete event simulation for spatially explicit problems. In *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, pages 46–53, 1998.

[26] DIAPM. *RTAI Programming Guide 1.0.* Politecnico di Milano - Dipartimento di Ingegneria Aerospaziale, 2000.

[27] Ernesto Kofman Federico Bergero. Integración del simulador PowerDEVS con el entorno Scilab. In *RPIC 2009 - XIII Workshop on Information Processing and Control*, Rosario, Argentina, 2009.

[28] Joaquín Fernández and Ernesto Kofman. Implementación autónoma de métodos de integración numérica QSS. Technical report, FCEIA - UNR, Rosario, Argentina, 2012.

[29] Xenofon Floros, Federico Bergero, Francois E. Cellier, and Ernesto Kofman. Automated Simulation of Modelica Models with QSS Methods - The Discontinuous Case. In *8th International Modelica Conference*, March 2011.

[30] Xenofon Floros, François E. Cellier, and Ernesto Kofman. Discretizing Time or States? A Comparative Study between DASSL and QSS. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT, Oslo, Norway, October 3, 2010*, pages 107–115, 2010.

[31] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[32] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-Interscience, New York, 2004.

[33] Peter Fritzson, Peter Aronsson, Hakan Lundvall, Kaj Nystrom, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Development Environment. *Proceedings of the 46th Conference on Simulation and Modeling (SIMS'05)*, pages 83–90, 2005.

[34] Peter Fritzson and Peter Bunus. Modelica - A General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation. In *Annual Simulation Symposium*, pages 365–380, 2002.

[35] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. In *ECOOP*, pages 67–90, 1998.

[36] M. Galassi. *GNU Scientific Library Reference Manual*, third edition, 2009.

[37] A. Gasparrini, P. Codoni, J. Danelón, and E. Kofman. Diseño, Implementación y Utilización de un Conversor A/D – D/A Asincrónico. In *Proceedings of AADECA 2010*, Buenos Aires, Argentina, 2010.

[38] Philippe Gerum. Xenomai-Implementing a RTOS emulation framework on GNU/Linux. Technical report, Philippe Gerum, 2004.

[39] D.W. Glazer and C. Tropper. On process migration and load balancing in Time Warp. *Parallel and Distributed Systems, IEEE Transactions on*, 4(3):318–327, 1993.

[40] Claude Ed. Gomez. *Engineering and scientific computing with Scilab.* Birkhäuser, Boston, 1999.

[41] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, June 1987.

[42] P. Heidelberger and D. M. Nicol. Conservative Parallel Simulation of Continuous Time Markov Chains Using Uniformization. *IEEE Trans. Parallel Distrib. Syst.*, 4(8):906–921, August 1993.

[43] S.G. Henderson. Input model uncertainty: why do we care and what should we do about it? In *Simulation Conference, 2003. Proceedings of the 2003 Winter*, volume 1, pages 90 – 100 Vol.1, dec. 2003.

[44] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

[45] Joon Sung Hong, Hae-Sang Song, Tag Gon Kim, and Kyu Ho Park. A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development. *Discrete Event Dynamic Systems*, 7(4):355–375, 1997. 10.1023/A:1008262409521.

[46] Shafagh Jafer and Gabriel Wainer. Conservative DEVS: a novel protocol for parallel conservative simulation of DEVS and Cell-DEVS models. In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, pages 140:1–140:8, San Diego, CA, USA, 2010. Society for Computer Simulation International.

[47] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[48] Monageng Kgwadi, Hui Shang, and Gabriel Wainer. Definition of dynamic DEVS models: Dynamic Structure CD++. In *Proceedings of the 2008 Spring simulation multiconference*, SpringSim '08, pages 10:1–10:4, San Diego, CA, USA, 2008. Society for Computer Simulation International.

[49] Ki Hyung Kim, Yeong Rak Seong, Tag Gon Kim, and Kyu Ho Park. Distributed simulation of hierarchical DEVS models: hierarchical scheduling locally and time warp globally. *Transactions of the Society for Computer Simulation International*, 13(3):135–154, 1996.

[50] Harold Klee. *Simulation of Dynamic Systems with MATLAB and Simulink*. CRC, 2007.

[51] E. Kofman. Relative Error Control in Quantization Based Integration. *Latin American Applied Research*, 39(3):231–238, 2009.

[52] Ernesto Kofman. A Second-Order Approximation for DEVS Simulation of Continuous Systems. *Simulation*, 78(2):76–89, 2002.

[53] Ernesto Kofman. Discrete Event Simulation of Hybrid Systems. *SIAM Journal on Scientific Computing*, 25:1771–1797, 2004.

[54] Ernesto Kofman. A Third Order Discrete Event Simulation Method for Continuous System Simulation. *Latin American Applied Research*, 36(2):101–108, 2006.

[55] Ernesto Kofman and Sergio Junco. Quantized-state systems: a DEVS Approach for continuous system simulation. *Trans. Soc. Comput. Simul. Int.*, 18(3):123–132, 2001.

[56] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, July 2003.

[57] Jason Liu. *Parallel Discrete-Event Simulation.* John Wiley & Sons, Inc., 2010.

[58] Qi Liu. *Algorithms for Parallel Simulation of Large-Scale DEVS and Cell-DEVS Models.* PhD thesis, Systems and Computer Engineering Dep. Carleton University, 2010.

[59] Real Time Engineers ltd. FreeRTOS website. http://www.freertos.org/.

[60] Martina Maggio, Kristian Stavåker, Filippo Donida, Francesco Casella, and Peter Fritzson. Parallel Simulation of Equation-based Object-Oriented Models with Quantized State Systems on a GPU. In *Proceedings of 7th Modelica Conference*, Como, Italy, 2009.

[61] Martina Maggio, Kristian Stavåker, Filippo Donida, Francesco Casella, and Peter Fritzson. Parallel Simulation of Equation-based Object-Oriented Models with Quantized State Systems on a GPU. In *Proceedings of 7th Modelica Conference*, Como, Italy, 2009.

[62] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. Technical report, Intel Technology Journal, 2002.

[63] Gustavo Migoni. *Simulación por Cuantificación de Sistemas Stiff.* PhD thesis, Universidad Nacional de Rosario, 2010.

[64] Gustavo Migoni and Ernesto Kofman. Linearly Implicit Discrete Event Methods for Stiff ODEs. *Latin American Applied Research*, 39(3):245–254, 2009.

[65] Jayadev Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.

[66] Mohammad Moallemi, Gabriel Wainer, Federico Bergero, and Rodrigo Castro. Component-Oriented Interoperation of Real-Time DEVS Engines. In *44th Annual Simulation Symposium*, Boston, MA, USA, April 2011.

[67] Stephen P. Molner. The Art of Molecular Dynamics Simulation. *Journal of Chemical Education*, 76(2):171, 1999.

[68] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[69] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989.

[70] R. Nikoukhah and S. Steer. SCICOS-A dynamic system builder and simulator. In *Computer-Aided Control System Design, 1996., Proceedings of the 1996 IEEE International Symposium on*, pages 430 –435, sep 1996.

[71] James Nutaro. *Parallel discrete event simulation with application to continuous systems*. PhD thesis, University of Arizona, 2003. AAI3119971.

[72] James Nutaro. A discrete event method for wave simulation. *ACM Trans. Model. Comput. Simul.*, 16(2):174–195, 2006.

[73] James J. Nutaro. *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Wiley Publishing, 2010.

[74] M. Otter, H. Elmqvist, and F. E. Cellier. Modeling of multibody systems with the object-oriented modeling language Dymola. *Nonlinear Dynamics*, 9:91–112, 1996. 10.1007/BF01833295.

[75] Cristian Perfumo, Ernesto Kofman, Julio Braslavsky, and John K. Ward. Load management: Model-based control of aggregate power for populations of thermostatically controlled loads. *Energy Conversion and Management*, 55:36–48, 2012.

[76] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 184 –592 Vol. 1, feb. 2005.

[77] PowerDEVS site at SourceForge - http://sourceforge.net/projects/powerdevs/.

[78] H. Praehofer and G. Reisinger. Distributed simulation of DEVS-based multiformalism models. In *AI, Simulation, and Planning in High Autonomy Systems, 1994. Distributed Interactive Simulation Environments., Proceedings of the Fifth Annual Conference on*, pages 150–156, 1994.

[79] D.M. Rao, N.V. Thondugulam, R. Radhakrishnan, and P.A. Wilsey. Unsynchronized parallel discrete event simulation. In *Simulation Conference Proceedings, 1998. Winter*, volume 2, pages 1563–1570, 1998.

[80] V. Savcenco and R.M.M. Mattheij. A Multirate Time Stepping Strategy for Stiff Ordinary Differential Equations. *BIT Numerical Mathematics*, 47:137–155, 2007.

[81] Lee Schruben. Simulation modeling with event graphs. *Commun. ACM*, 26(11):957–963, November 1983.

[82] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[83] Tapas K. Som and Robert G. Sargent. Model structure and load balancing in optimistic parallel discrete event simulation. In *Proceedings of the fourteenth workshop on Parallel and distributed simulation*, PADS '00, pages 147–154, Washington, DC, USA, 2000. IEEE Computer Society.

[84] Kristian Stavaker. *Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units*. PhD thesis, Linkoping Studies in Science and Technology - Linkoping, Sweden, 2011.

[85] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[86] Y. Tang, K.S. Perumalla, R.M. Fujimoto, H. Karimabadi, J. Driscoll, and Y. Omelchenko. Optimistic parallel discrete event simulations of physical systems using reverse computation. In *Principles of Advanced and Distributed Simulation, 2005. PADS 2005. Workshop on*, pages 26–35, 2005.

[87] H. Vangheluwe. DEVS as a Common Denominator for Multi-formalism Hybrid Systems Modelling. In *IEEE International Symposium on Computer Aided Control System Design*, pages 129–134, Anchorage, Alaska, 2000.

[88] Tim P. Vogels and L. F. Abbott. Signal Propagation and Logic Gating in Networks of Integrate-and-Fire Neurons. *The Journal of Neuroscience*, 25(46):10786–10795, 2005.

[89] Gabriel Wainer. CD++: a toolkit to develop DEVS models. *Software: Practice and Experience*, 32(13):1261–1306, 2002.

[90] Gabriel Wainer, Gastón Christen, and Alejandro Dobniewski. Defining DEVS Models with the CD++ Toolkit. In *Proceedings of ESS2001*, pages 633–637, Marseille, France, 2001.

[91] Gabriel A. Wainer. *Discrete-Event Modeling and Simulation: a Practitioner's approach*. CRC Press. Taylor and Francis, 2009.

[92] WindRiver. VxWorks website. http://www.windriver.com/.

[93] B. Zeigler and S. Vahie. Devs formalism and methodology: unity of conception/diversity of application. In *Proceedings of the 25th Winter Simulation Conference*, pages 573–579, Los Angeles, CA, 1993.

[94] Bernard Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation - Second Edition*. Academic Press, 2000.

[95] Bernard P. Zeigler and J. S. Lee. Theory of Quantized Systems: Formal Basis for DEVS/HLA Distributed Simulation Environment. *Enabling Technology for Simulation Science II*, 3369(1):49–58, 1998.