Conversión de modelos PowerDEVS al lenguaje Modelica

Tesina de grado para la obtencion del grado de Licenciado en Ciencias de la Computación

Tesinista :
Luciano Andrade

Director :
Federico Bergero

Co-Director:
Ernesto Kofman

28 de septiembre de 2015

Índice general

Resumen

El modelado y simulación se han convertido en una actividad centrales de todas las disciplina ingenieriles y científicas, son utilizados en el análisis de sistemas ayudándonos a ganar un mejor entendimiento de su funcionamiento. Son importantes para el diseño de nuevos sistemas donde podemos predecir el comportamiento del sistema antes de que sea construido. El modelado y simulación son las únicas técnicas disponibles que nos permiten analizar sistemas arbitrarios no lineales bajo una variedad de condiciones experimentales.

PowerDEVS es un entorno integrado para el modelado y simulación basada en el formalismo DEVS, permite definir modelos atómico en C++ que puede ser conectados gráficamente en bloques jerárquicos para crear sistemas más complejos. El entorno automáticamente transforma el modelo a código C++ que ejecuta la simulación.

QSS-Solver es una implementación de los métodos de integración por cuantificación de estados (Quantized State System) los cuales permiten, a diferencia de las implementaciones sobre eventos discretos no desperdician carga computacional en la simulación de eventos discretos.

En este trabajo se describe la implementación de una aplicación capaz de convertir modelos descriptos en el entorno PowerDEVS, a modelos en el lenguaje Modelica, más especificamente en μ-Modelica. Permitendo al usuario describir modelos de forma gráfica de diagrama de bloques, lo cual resulta más ameno para nuevos usuarios y aquellos que no se encuentran familiarizados con la programación en general, además de permtirle al usuario acceder a gran cantidad de modelos ya desarrollados en esta herramienta.

Esta herramienta convierte los modelos (descriptos en PowerDEVS) en código Modelica, más especificamente código μ-Modelica (un subconjunto de Modelica), permitiendo ejecutar este modelo (convertido) con alguno de los compiladores Modelica, OpenModelica o Dymola, o "QSS Solver", lo cual nos permite ganar al menos un orden de magnitud en los tiempos incurridos en la simulación.

Capítulo 1

Introducción

1.1. Motivación

El modelado y simulación[**Zeigler**] se han convertido en una actividad centrales de todas las disciplina ingenieriles y científicas, son utilizados en el análisis de sistemas ayudándonos a ganar un mejor entendimiento de su funcionamiento.

Son importantes para el diseño de nuevos sistemas donde podemos predecir el comportamiento del sistema antes de que sea construido. El modelado y simulación son las únicas técnicas disponibles que nos permiten analizar sistemas arbitrarios no lineales bajo una variedad de condiciones experimentales.

Veamos algunas razones por las cuales utilización de simulaciones es deseable o incluso requerido:

- El sistema físico no se encuentra disponible, se debido a que el sistema no fue aun construido o si el sistema debería ser construido.
- El experimento puede ser peligroso. Usualmente, simulaciones son realizadas para determinar si el experimento real "explotara", poniendo al experimentador en peligro.
- El costo del experimento es demasiado alto o las herramientas necesarias no se encuentran disponibles o son muy costosas. Tambien es posible que el sistema se encuentra siendo utilizado y tomar el tiempo para experimentar sería inaceptable.
- Los tiempos del sistema no son compatibles con el del experimentador. Usualmente simulaciones son utilizadas debido a que el experimento real se realiza tan rapido que no es posible observarlo (por ejemplo una explosión) o porque el experimento toma tanto tiempo que el experimmentador estaria muerto cuando el experimento se encuentre completado.
- Variables de control, de estado y/o del sistema pueden encontrarse inaccesibles. Usualmente simulaciones son utilizadas debido a que nos permite acceder todas las variables de entrada y todos los estados, mientras que el sistema real, algunas entradas (ruidos, por ejemplo) no son manipulables y algunas variables internas del sistema no son accesibles a la medición. Simulaciones tambien nos permite manipular el modelo en formas que no podriamos manipular el sistema real, por ejemplo, podemos decidir cambiar la masa de un objeto de 50 kg a 400 kg y repetir la simulación. En un sistema físico, la modificación anterior es imposible o requiere una costoza y larga alteración del sistema.
- Eliminación de perturbaciones. Usualmente, se llevan adelante simulaciones que nos permite eliminar perturbaciones que son inevitables en

el sistema real. Lo que nos permite aislar efectos particulares, y puede conducir a mejores apreciaciones sobre el comportamiento general del sistema.

- Eliminación de efectos de segundo orden. Usualmente, se utilizan simulaciones porque nos permite eliminar efectos de segundo orden (como no linealidades de componentes del sistema). Nuevamente esto ayuda a obtener un mejor entendimiento del comportamiento general del sistema.

Es por esto que cuando corremos un modelo es deseable que pueda ser simulado de la forma más rapida y eficiente posible.

Para realizar la simulación debemos generar el modelo, es decir, la descripción de nuestro sistema de forma que sea posible compilarse en código de maquina para poder ser ejecutado (pasando por un lenguaje de propósito general, usualmente C o C++).

El modelo inicia como una función matemática de la forma

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

donde x representan las variables del sistema, u el estado inicial y t el tiempo, este modelo, puede ser convertido en un modelo Modelica[**Fri98**][**Fritzson02modelica**] de forma textual o gráfica, dependiendo de las herramientas con las que contemos y como nos resulte más simple de describir.

PowerDEVS[**BK11**] es una herramienta de simulación de sistemas híbridos, basado en el formalismo DEVS[**Zeigler**], con una interfaz gráfica orientada a bloques, donde los bloques pueden ser conectados entre si, modificado sus parámetros, además permite conectarse con el entorno Scilab para poder utilizar expresiones y herramientas de cálculo provistas por este entorno.

Nos interesa poder utilizar el entorno PowerDEVS[**BK11**], debido a que no solo la interfaz gráfica es más amena para usuarios que no estan necesariamente habituados a la programación, sino que tambien deseamos utilizar los modelos ya definidos en esta herramienta.

En este aspecto, contamos con la herramienta "QSS-Solver"[**Ber12**], la cual nos permitiria ejectura simulaciones un orden de mágnitud más rápido, que otras implementaciones.

Por lo cual enn este trabajo nos proponemos mostrar una aplicación capaz de convertir modelos descriptos en la herramienta PowerDEVS[**BK11**] a modelos en el lenguaje Modelica[**Fri98**][**Fritzson02modelica**], más específicamente en μModelica[**Ber12**], capaz de ser ejecutados en el QSS-Solver[**Ber12**], obteniendo lo mejor de los dos mundos.

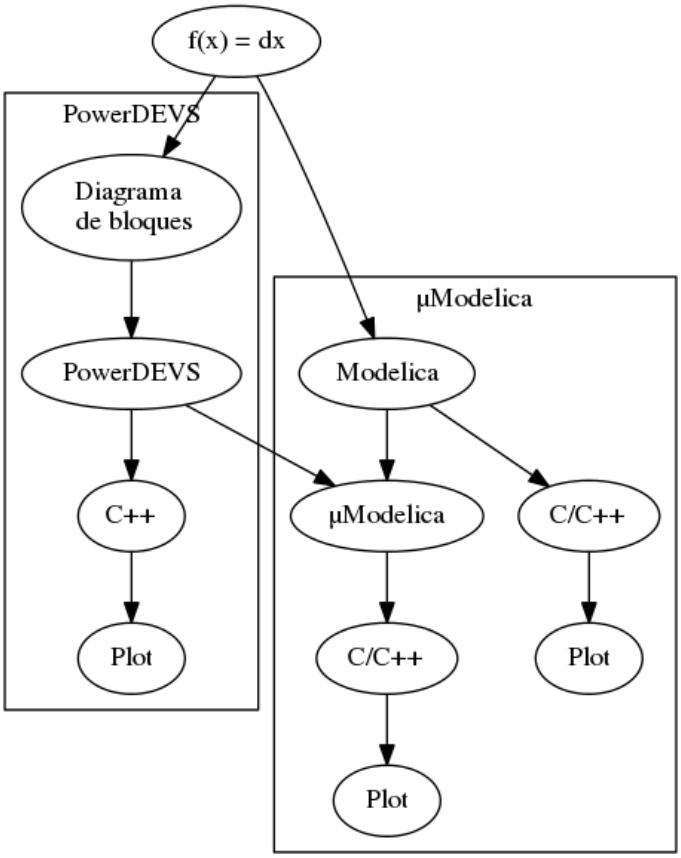


Figura 1.1: Esquema de conversiones

En la figura 1.1, se muestran los dos principales estrategias (PowerDEVS[**BK11**] y Modelica[**Fri98**]) para realizar una simulación. En el caso de PowerDEVS, el primer paso es convertir el sistema en diagramas de bloques, luego en DEVS[**Zeigler**], en PowerDEVS el cual puede automáticamente convertirlo en C++ y luego obtener los resultados ejecutando este modelo.

Desde la perspectiva de Modelica (o μ-Modelica), debemos pasar el sistema a Modelica (o μ-Modelica) y luego el compilador se encargara de generar código (usualmente C o C++) capaz de correr la simulación y obtener resultados.

El actual trabajo está representado por la flecha que sale de PowerDEVS hacia μmodelica, permitiendo especificar la simulación en diagramas de bloques y ejecutar la simulación en el QSS-Solver, en el lenguaje μmodelica.

1.2. Organización del trabajo

El presente trabajo se organiza en 6 capítulos:

- *Introducción* en la cual ya vimos una visión general del objetivo así como las herramientas que utilizaremos y algunas herramientas relacionadas
- *Conceptos previos* en este capítulo veremos los fundamentos matemáticos y profundizaremos sobre las dos herramientas principales que conciernen este trabajo.
- *Conversión de modelos DEVS* en este capítulo veremos en detalles la conversión de un modelo desde su formulación matemática, en su modelo en PowerDEVS y veremos las conversiones necesarias para llevar adelante la conversión a μ-Modelica.
- *Detalles de la implementación* en este capítulo se muestra los componentes de software que forman la aplicación, y descripción en pseudocódigo de los principales componentes.
- *Ejemplos y Resultados* en este capítulo vemos varios ejemplos y realizamos una comparación de los resultados obtenidos a partir de compara los tiempos de simulaciones de los modelos originales, en PowerDEVS, y los modelos convertidos en μ-Modelica.
- *Conclusiones*, por último revisamos nuestras conclusiones y proponemos trabajos a futuro.

1.3. Trabajo relacionado

En [**Ber12**] se describe una extensión del Compilador OpenModelica el cual traslada modelos regulares Modelica a un subconjunto más simple μ-Modelica, el cual puede ser interpretado directamente por el QSS-Solver.

ModelicaDEVS [**Beltrame06quantisedstate**] es una librería Modelica que permite describir simulaciones DEVS, ofrece una re-implementación de PowerDEVS dentro del marco de Modelica.

DESlib [**Sanz09paralleldevs**] es una librería para la descripción de modelos Parallel DEVS y Modelado orientado a proceso en Modelica. La librería contiene cuatro paquetes que pueden ser utilizados para modelar sistemas de eventos discretos:

- RandomLib puede generar números y variables aleatorias, siguiendo distribuciones de probabilidades discretas y continuas.
- DEVSLib puede ser utilizado para modelar sistemas de eventos discretos (DEVS) siguiendo el formalismo de parallel DEVS.
- SIMANLib y ARENALib puede ser utilizado para modelar sistemas de eventos discretos (DEVS) siguiendo el enfoque orientado al proceso.

Estas dos librerías llevan los formalismos DEVS hacia Modelica, pero no utilizan la herramienta powerdevs, ni se ejecutan en QSS-Solver. Podríamos desarrollar los modelos utilizando estas librerías y ejecutar la simulación del modelo convertido en μ-Modelica con el QSS-Solver, pero esto no permite re-utilizar los modelos desarrollados en PowerDEVS.

M/CD++ [**conf/mascots/DAbreuW05**] es una herramienta para convertir simulaciones en un subconjunto de Modelica, a simulaciones DEVS, este trabajo funciona en sentido opuesto a nuestro trabajo, es decir convirtiendo modelos Modelica en modelos DEVS, por lo que no utiliza PowerDEVS y no ejecuta la simulación en el QSS-Solver[**Ber12**].

Capítulo 2

Conceptos Previos

En este capítulo, basado en [**Fer12**], [**Ber12Th**], [**BK11**], [**BK13**], introducimos algunos conceptos básicos, necesarios para poder comprender este trabajo, modelado y los formalismos y simulación de sistemas y clasificación.

2.1. Modelado y Simulación

El Modelado y Simulación[**Zeigler**] de un Sistema es el proceso por el cual se desarrolla un modelo, el cual es luego ejecutado, de forma de obtener datos sobre el comportamiento del sistema. El modelo debe conservar las principales características del sistema, pero al mismo tiempo ser significativamente más simple, de forma que al momento de simularlo sea más eficaz utilizar la simulación que el sistema en sí.

2.1.1. Sistemas Continuos y Discretos

Se considera un sistema continuo si las variables de éste son conocidas en cada instante de tiempo, mientras que se considera discreto si las variables son conocidas en instantes de tiempo determinados.

En general los sistemas en estudio serán continuos, pero deberemos utilizar sistemas discretos puésto que la simulación en computadora así lo requiere, dado que la misma computadora es un sistema discreto.

2.1.2. Métodos de Integración numérica

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

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

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

$$x(t = t_0) = x_0 \tag{2.2}$$

Sea $x_i(t)$ la trayectoria del estado i-esimo 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. Esta 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:

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

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

$$x_i(t^* + h) = x_i(t^*) + f_i(t^*) \cdot h + \frac{d^2x_i(t^*)}{dt^2} \cdot \frac{h^2}{2!} + \cdots \tag{2.4}$$

Los distintos algoritmos de integración difieren en la manera de aproximar las derivadas superiores de f y en el número de términos de la serie de Taylor que consideran para la aproximación, entre los principales métodos podemos mencionar Runge-Kutta, DASSL, DOPRI, etc.

A modo de ejemplos introducimos un ejemplo el cual desarrollamos desde distintas herramientas el sistema de ecuaciones de Lotka-Volterra, también conocidas como ecuaciones predador-presa o presa-predador, son un par de ecuaciones diferenciales de primer orden no lineales que se usan para describir dinámicas de sistemas biológicos en el que dos especies interactúan, una como presa y otra como depredador.

$$\frac{dx}{dt} = x(\alpha - \beta y)$$
$$\frac{dy}{dt} = -y(\gamma - \delta x)$$

donde:

- y es el número de algún predador (por ejemplo, un lobo)
- x es el número de sus presas (por ejemplo, conejos)
- $\frac{dy}{dt}$ y $\frac{dx}{dt}$ representa el crecimiento de las dos poblaciones en el tiempo
- t representa el tiempo y
- α, β, γ y δ son parámetros que representan las interacciones de las dos especies.

2.2. Modelica

Modelica[**Fri98**][**Fritzson02modelica**] es un lenguaje orientado a objetos desarrollado para describir de manera sencilla modelos de sistemas dinámicos eventualmente muy complejos. Además de las características básicas de todo lenguaje orientado a objetos, contiene herramientas específicas que permiten describir las relaciones constitutivas de los distintos componentes de cada modelo y las relaciones estructurales que definen la interacción entre dichos componentes. De esta manera, el lenguaje permite asociar cada

componente de un sistema a una instancia de una clase. Adicionalmente, los componentes típicos de los sistemas de distintos dominios de la física y de la técnica pueden agruparse en librerías de clases para ser reutilizados. De hecho, existe una librería estándar de clases de Modelica, que contiene los principales componentes básicos de sistemas eléctricos, mecánicos (traslacionales, rotacionales y multicuerpos), térmicos, state graphs, y diagramas de bloques. Otras librerías (disponibles en la web) contienen componentes de sistemas hidráulicos, bond graphs, redes de petri, etc. Por otro lado, las herramientas que provee Modelica para expresar relaciones estructurales de un modelo permiten construir la estructura del mismo de una manera totalmente gráafica, lo que a su vez permite describir un sistema mediante un diagrama muy similar al del Sistema Físico Idealizado. Como con todo lenguaje, para poder simular un modelo descripto en Modelica es necesario utilizar un compilador. Actualmente existen tres compiladores más o menos completos de Modelica: Dymola, MathModelica y OpenModelica. Los dos primeros son herramientas comerciales que cuentan con interfaces gráficas para construir los modelos. OpenModelica es una herramienta libre, de código abierto.

```
1  class LotkaVolterra
2    Real x(start = 0.5);
3    Real y(start = 0.5);
4    parameter Real a = 0.1;
5    parameter Real b = 0.1;
6    parameter Real c = 0.1;
7    parameter Real d = 0.1;
8  equation
9    der(x) = x * (a - b * y);
10   der(y) = - y * (d - c * x);
11 end LotkaVolterra;
```

Listado 1: LotkaVolterra.mo

Continuando con el ejemplo del sistema de ecuaciones de Lotka-Volterra, en el listado 2 se muestra el equivalente modelo en Modelica, en el se pueden observar algunas particularidades del lenguaje.

En la primera linea se puede ver que el modelo, en este caso una "clase", y su nombre LotkaVolterra. Modelica, utiliza 7 clases restringidas: **block**, **connector**, **function**, **model**, **package**, **record** y **type**. Cada una de estas clases restringidas permite declarar clases más específicas cualquiera de ellas puede reemplazarse por **class**, pero siempre es mejor especificar de que tipo de clase se trata para mejorar la legibilidad y facilitar la depuración de código.

Las siguientes dos lineas declaran las variables `x` e `y` del tipo `Real`, ambas

con valor de inicio `0.5`, las lineas 4 a 7, declaran los parámetros `a`, `b`, `c` y `d` a diferencia de `x` e `y`, estos parámetros no pueden cambiar durante la simulación, pero pueden cambiar de simulación en simulación.

Esta primera parte (lineas 2 a 7) del modelo constituye la sección de declaraciones, se encuentra la sección de ecuaciones (luego de la palabra clave `equation`), esta sección describe el sistema de ecuaciones de nuestro modelo. Donde la expresión `der(x)` representa la derivada `x` respecto del tiempo y análogamente con `y`. Es importante notar que el símbolo `=` no es el símbolo de asignación, sino de igualdad matemática (es decir acausal), a diferencia de la mayoría de los lenguajes de programación modernos. Para realizar asignaciones se utiliza una sección `algorithm`, que contiene asignaciones ordenadas. Con el fin de distinguir de las ecuaciones de la sección `equation`, se utiliza el operador `:=`, varias asignaciones puede ser ejecutadas en la sección de `algorithm`. Ademas la sección `algorithm` puede tener expresiones if-then-else, construcciones if-then-else, y bucles.

Por supuesto Modelica es un lenguaje mucho más extenso de lo que hemos descripto, pero los conceptos que utilizamos a lo largo de este trabajo son los expresados en la presente sección.

2.3. Métodos de integración QSS

Los métodos Quantized State System [**Fer12**], [**Ber12**], [**Beltrame06quantisedstate**], [**Cel06**], pueden aproximar Ecuaciones Diferenciales Ordinarias (ODE por sus siglas en inglés) mediante modelos de eventos discretos cuantificando los estados del sistema, a diferenencia de los métodos mencionados en la sección 2.1.2, los cuales realizan la integración sobre el timpo.

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

$$\dot{x}(t) = f(q(t), v(t)) \tag{2.5}$$

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

$$q_j(t) = \begin{cases} x_j(t) & \text{si} \mid x_j(t) - q_j(t^-) \mid \geq \Delta Q_j \\ q_j(t^-) & \text{en caso contrario} \end{cases} \tag{2.6}$$

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

En la Figura 2.1 vemos la relación entrada-salida de una función de cuantificación de orden cero.

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 $v(t)$ son aproximaciones constantes a trozos de las componentes correspondientes de $u(t)$. Los

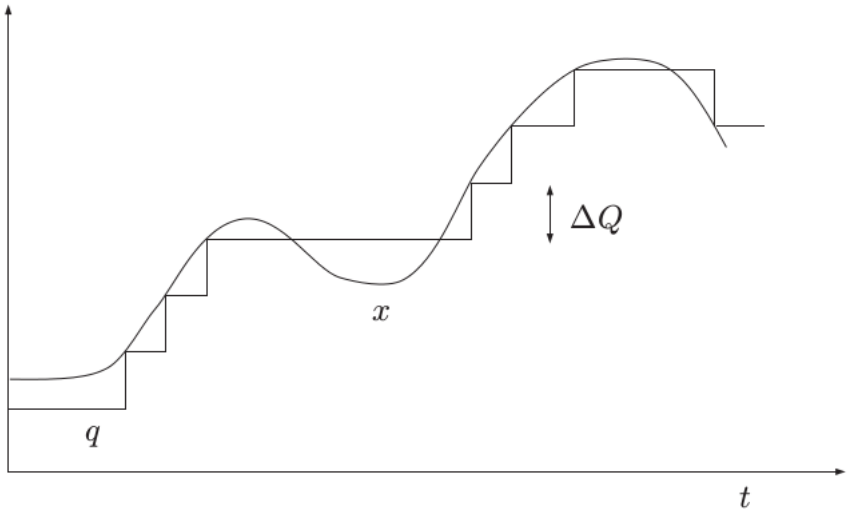


Figura 2.1: Comportamiento de un modelo DEVS atómico

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. Dependiendo del orden del método, las variables de estado siguen trayectorias lineal a trozos, parabólica a trozos o constante a trozos, 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.

2.4. μ-Modelica

El lenguaje μ-Modelica[**Ber12**] tiene las siguientes restricciones con respecto a Modelica:

- El modelo es plano, es decir no permite clases.
- Todas las variables pertenecen al tipo predefinido Real y solo hay tres categorías de variables: estado continuo, estado discreto y variables algebraicas.
- Los parámetros también son de tipo Real.
- Arreglos están permitidos. Indices en los arreglos dentro de cláusulas `for` están restringidos a la forma $\alpha \cdot i + \beta$, donde α y β son expresiones enteras y `i` es el índice de la iteración.
- La sección de ecuaciones está compuesta de :
  - Definición de variables de estados : $der(x) = f(x(t), d, a(t), t)$; ODE en forma explícita
  - Definición algebraica : $(a_1, \ldots, a_n) = g(x(t), d, a(t), t)$;

  con la restricción de que cada variable algebraica solo puede depender del estado y de variables algebraicas previamente definidas.
- Discontinuidades son expresadas solo con las clausulas *when* y *elsewhen* dentro de la sección *algorithm*. Las condiciones dentro de las dos clausulas solo pueden ser relaciones ($<$, $\leqslant$, $>$ $\geqslant$) y, dentro de la clausula, solo asignaciones de variables discretas y *reinit* de estados continuos son permitidos

2.5. Stand–Alone QSS solver

Como mencionamos antes los métodos QSS de integración remplazan la discretización del tiempo de los métodos clásicos por una quantificación de las variables del sistema. De esta forma, estos métodos generan aproximaciones del sistema continuo y tienen algunas ventajas sobre sus contra partes clásicas.

La forma más simple de implementar algoritmos QSS es mediante el uso de un simulador DEVS, de hecho PowerDEVS implementa la totalidad de la familia de algoritmos QSS. Estas implementaciones aunque simples, son ineficientes, pues desperdician mucho poder computacional en sincronizar y transmisión de eventos.

Estas desventajas motivo el desarrollo del *Stand–Alone QSS solver*[**Ber12**], implementados como un conjunto de módulos en lenguaje C. Este implementa toda la familia de métodos QSS y permite que los modelos contengan discontinuidades de tiempo y estado.

Una dificultad impuesta por los métodos QSS es que hace uso de información estructural del modelo. Cada paso en un método QSS involucra un cambio en una variable de estado y en la derivada del estado que depende de el. Por lo que el modelo debe proveer no solo la expresión para calcular las derivadas del estado (como en un clásico ODE solver) pero además un matriz de incidencias para informar al solver que derivadas de estado han cambiado luego de cada paso.

Como sería muy incomodo para el usuario proveer esta información estructural, el solver tiene una interfaz que automáticamente obtiene la matriz de incidencia desde una definición estándar de modelos.

La interfaz permite al usuario describir el modelo utilizando (μ-Modelica) y automáticamente genera el código C del modelo incluyendo la estructura.

Continuando con nuestro ejemplo sobre el sistema Lotka Volterra, incluimos el modelo proporcionado en la instalación del QSS-Solver, este es equivalente al modelo anterior, excepto que la variable `x` es un arreglo de dos dimensiones juando el papel de `x` e `y`. Tambien se puede apreciar un bloque extra `initial algorithm` el cual inicializa las variables (en lugar de utilizar la inicialización estandar).

```
model lotka_volterra
        Real x[2];
        initial algorithm
                x[1] := 0.5;
                x[2] := 0.5;
        equation
                der(x[1]) = 0.1 * x[1] - 0.1 * x[1]*x[2];
                der(x[2]) = 0.1 * x[1]*x[2] - 0.1 * x[2];
        annotation(

        experiment(
                MMO_Description="Lotka Volterra model",
                MMO_Solver=QSS3,
                MMO_Output={x[:]},
                StartTime= 0.0,
                StopTime= 300.0,
                Tolerance={ 1e-3},
                AbsTolerance={ 1e-6}
        ));
end lotka_volterra;
```

Listado 2: LotkaVolterra.mo

2.6. Formalismo DEVS

DEVS[**Zeigler**] 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.

2.6.1. 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:

$$(X, Y, S, \delta_{int}, \delta_{ext}, \lambda, t_a) \text{ donde :} \tag{2.7}$$

- 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 $S \rightarrow \mathbb{R}_0^+$, que indica cuánto tiempo el modelo atómico permanecerá en un estado dado, si es que no se recibe ningún evento de entrada. Esta función es *Función de Avance de Tiempo.*
- δ_{int} es una función $S \rightarrow S$, 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, es la *Función de Transición Interna.*
- δ_{ext} es una función $(S \times \mathcal{P}(\mathbb{R}_0^+) \times X) \rightarrow S$, que indica el cambio de estado ante la presencia de un evento externo, esta es la *Función de Transición Externa.*
- λ es una función $S \rightarrow Y$ que indica qué evento se debe emitir al salir de un estado dado, es *Función de Salida.*

Los conjuntos S, X e Y son arbitrarios, y en general infinitos. 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 2.2 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 $\lambda(\lambda : S \rightarrow Y)$ 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.

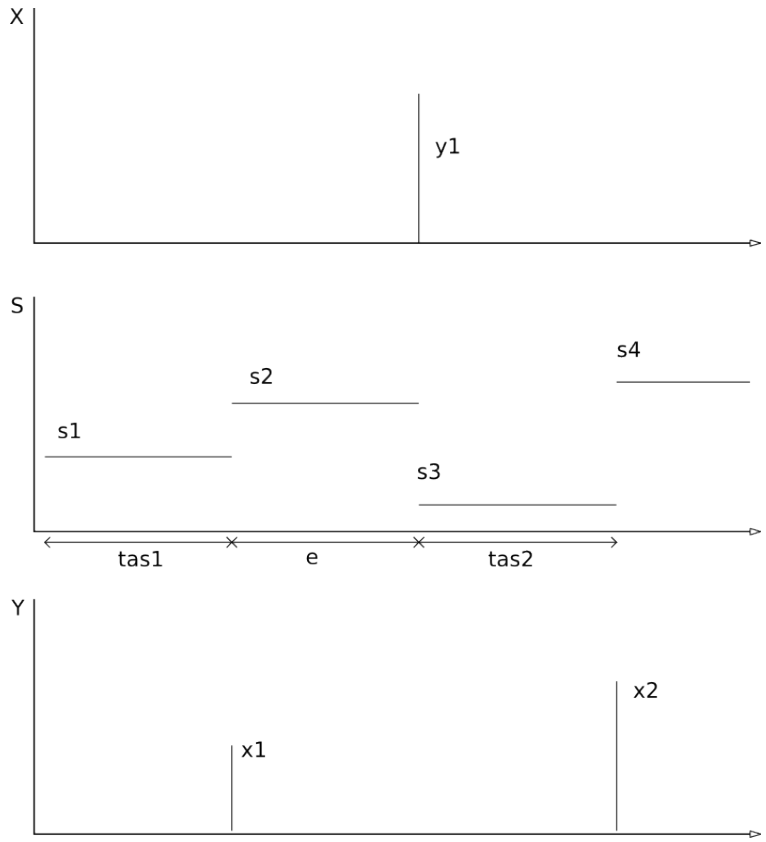


Figura 2.2: Comportamiento de un modelo DEVS atómico

2.6.2. 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, 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:

$$N = (X_N, Y_N, D, M_d, EIC, EOC, IC, Select) \tag{2.8}$$

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 son el conjuntos de conexiones entre los modelos internos y los puertos del modelo acoplado:
  - 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.
- IC representa las conexiones internas del modelo acoplado.
- Select es una función $(\mathcal{P}((D)) \rightarrow D)$ 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.

Formalmente:

$$EIC \in \{((N, ip_N), (d, ip_d))|ip_N \in InPorts, d \in D, ipd \in InPorts_d\}$$
$$EOC \in \{((d, op_d), (N, op_N))|op_N \in OutPorts, d \in D, opd \in OutPorts_d\}$$

donde N es el modelo acoplado.

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

donde no se permite que $a = b$.

$InPorts$ 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.

EIC, EOC y IC son conjunto de pares, de pares, como se encuentran conectados los modelos (atómicos o acoplados) a través de sus puertos con los puertos de entrada, salida y con otros modelos (atómicos o acoplados) respectivamente. Tanto los puertos como los modelos son señalados por números dentro de PowerDEVS, por lo que estos conjuntos estan comprendidos por elementos de la forma $(m_a, p_a), (m_b, p_b)$, donde m_a y m_b son modelos (atomicos o no) en el actual modelo y p_a y p_b son sus correspondientes puertos.

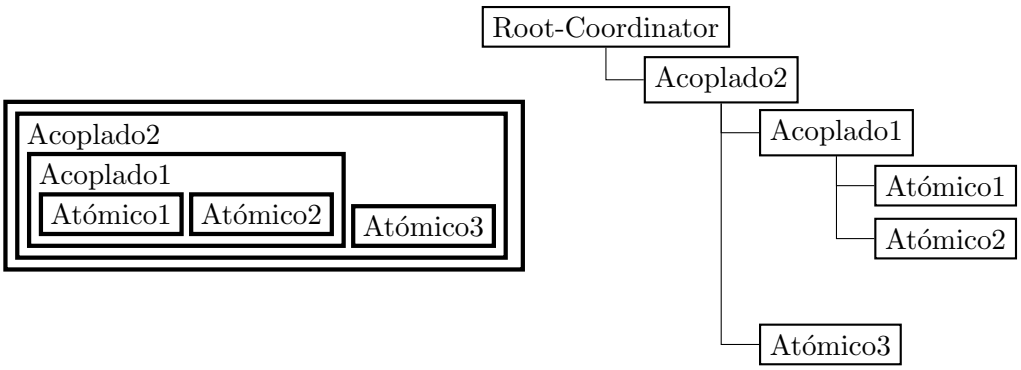


Figura 2.3: Ejemplo de un modelo jerarquico.

2.6.3. Modelos DEVS parametrizados

Definiremos primero los modelos DEVS parametrizados[**BKC12**] como un paso previo hacia el formalismo DEVS vectorial (Vectorial DEVS or VECDEVS), el cua es una herramienta que nos facilitará representar modelos de gran escala en forma gráfica, en particular este formalismo se encuentra implementado en la herramienta PowerDEVS.

Formalmente : dado un modelo DEVS atómico M obtenemos un Modelo DEVS Parametrizado:

$$M(p) = \{X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta, p\} \tag{2.9}$$

donde $p \in P$ es un parámetro que pertenece a un conjunto de parámetros arbitrario tal que δ_{int} , δ_{ext} , λ 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).

2.6.4. Modelos Vectoriales

Dado el modelo escalar DEVS Parametrizado:

$$M(p) = \{X, Y, S, \lambda_{int}, \lambda_{ext}, \lambda, ta, p\} \tag{2.10}$$

definimos un modelo Vectorial DEVS[**BKC12**] como la estructura:

$$V_D = \{N, X_V, Y_V, P, \{M_i\}\}, \tag{2.11}$$

donde:

- $N \in \mathbb{N}$ es la dimensión del modelo vectorial.
- $X_V = X \times Index \bigcup \{-1\}$ es el conjunto de eventos de entradas vectorial donde X es el conjunto de eventos de entrada del modelo escalar e $Index = 1, \ldots, N$ es el conjunto de índices que indican cuál de los modelos DEVS atómicos recibirá el evento.
- $Y_V = Y \times Index$ es el conjunto de eventos de salida vectorial donde Y es el conjunto de eventos de salida del modelo escalar e $Index = 1, \ldots, N$ es el conjunto de índices que indica que 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.

Interfaz entre DEVS Vectorial y DEVS

Para conectar bloques vectoriales y bloques escalares es necesarios bloques que hagan de interfaz entre los dos formalismo, además, introducimos un bloque necesario para realizar modelos más complejos y conectar los diferentes componentes de un modelo vectorial entre sí.

- Escalar a Vector (Scalar to Vector): Este boque simplemente agrega al indice i al evento escalar que recive, transformándolo en un evento vectorial. Este modelo también posee un comportamiento especial para enviar el mismo evento en todas las componentes vectorial al mismo tiempo, cuando $i = -1$, cada evento de entrada es trasmitido para todas las componentes del vector salida.

- Vector a escalar (Vector to Scalar): Este bloque tiene un parámetro i que contiene el índice del vector de eventos a retransmitir, cuando recibe un evento con indice $j = i$, remueve el indice y retransmite el evento escalar.

- Index Shift: El modelo más simple es el Index Shift. Cuando se recibe un evento con el valor (x, i), envia un evento de salida $(x, i+sh)$, donde sh es un parámetro entero.

Es importante ver que los mensajes entre bloques vectoriales es un par donde uno de sus componenetes es un número natural, que funciona como índice, el cual nos permite determinar en que posición del vector se encuentra la otra componente.

En este trabajo todos estos bloques se la ha agregado la dimensión N como parámetro, esto es necesario para realizar la conversión de los modelos, de forma que no existan desconexiones a nivel Modelica, lo cual se reflejaría en un modelo con menos ecuaciones que variables, es decir un modelo no balanceado.

2.7. PowerDEVS

PowerDEVS[**BK11**] es un programa, concebido para ser utilizado por expertos programadores DEVS, así como usuarios finales que solo quieren conectar bloques y simular.

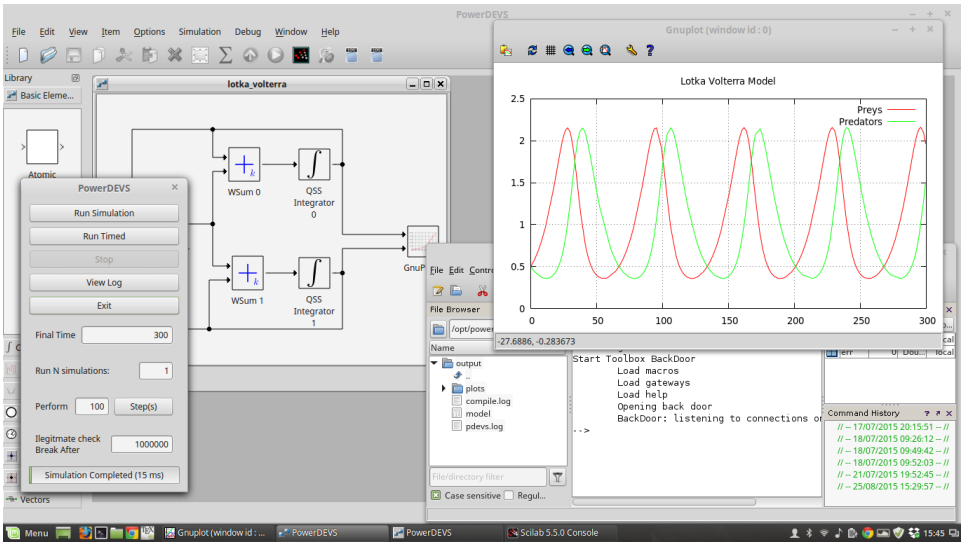


Figura 2.4: Interfaz gráfica de PowerDEVS

PowerDEVS esta compuesto por varios programas independientes:

- El *editor de modelos*, es desde el punto de vista del usuario, el principal programa de PowerDEVS, pues provee la interfaz gráfica y enlaces para las demás aplicaciones. Además de construir, manejar modelos y librerías, permite lanzar las simulaciones (lanzando el *Pre procesador*) y editar los bloques elementales hasta su definición atómica de modelo (invocando el *Editor Atómico*). La ventana principal de *Editor de Modelo* permite al usuario crear y abrir modelos y librerías. También permite a las librerías ser exploradas y los bloques arrastrados de las librerías a los modelos. La ventana del Modelo provee todas las funcionalidades típicas para la edición gráfica para poder copiar, cambiar el tamaño, rotar, etc. mientras que las conexiones pueden ser dibujadas entre los diferentes puertos. La ventana de Edición de Bloques, nos permite configurar la apariencia gráfica y elegir los parámetros del bloque y, en el caso de los modelos atómicos, seleccionar el archivo que contiene el código asociado con la definición DEVS.

- El *Editor de Modelos Atómicos* facilita la edición del código C++ correspondiente a cada modelo atómico DEVS, el usuario debe definir las variables que forman el estado y parametros y 6 funciones en sus correspondientes solapas, Init, Time Advance, Internal transition, External transition, Output y Exit Cuando el modelo se guarda, el código es guardado en los archivos .cpp y .h.

- El *Pre procesador*, toma un archivo .pdm (o .pds) producido por el *Editor de Modelos* y produce el programa que corre la simulación. Básicamente traduce el archivo .pdm a un archivo de cabecera .h que enlaza el simulador y el coordinador de acuerdo a la estructura del modelo pasando además los parámetros definidos para el modelo. El *Pre procesador*, además produce un makefile (Makefile.include) el cual invoca el compilador para generar el programa que implementa la simulación.

- La *interfaz de simulación*, que corre el programa que implementa la simulación y permite variar parámetros de la simulación como tiempo final, números de simulación a ejecutar, y el modo de simulación (normal, cronometrada, pasa o paso, etc.).

- Una instancia de Scilab, que actua como un espacio de trabajo, donde los parámetros pueden ser leídos, y los resultados pueden ser exportados.

En la figura 2.5 se puede ver el modelo Lotka Volterra que acompaña la instalación de PowerDEVS, este modelo (aunque no visible en la imagen) tiene valores iguales a los expresados en el modelo equivalente anteriormente presentado en el listado 2. las dos conecciones contra el bloque `GnuPlot 0` representando las variables `x` e `y` es decir presa y depredadores, ambas

variables son el resultado de la integración por lo que los integradores `QSS Integrato 0` y `QSS Integrato 1` possen a la entrada los valores `dx/dt` y `dy/dt`. Los bloques `WSum 0` y `WSum 1`, tiene constante de forma que el valor de salida (`y`) queda definido como $y = K[0] * u0 + K[1] * u1 + \cdots + K[7] * u7$ y en nuestro modelo son $K[0] = 0{,}1$ y $K[1] = -0{,}1$ en ambos bloques `WSum 0` y `WSum 1`. Replicando nuestra la equación de nuestro modelo.

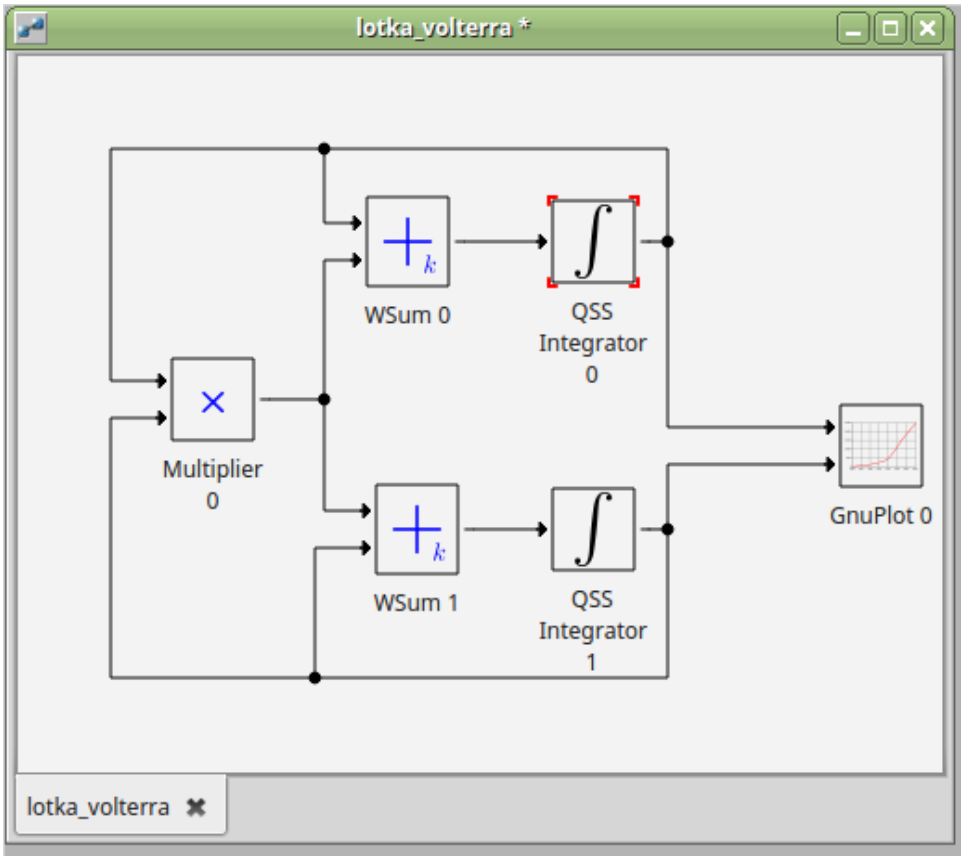


Figura 2.5: Modelo Lotka Volterra descripto en PowerDEVS

Capítulo 3

Conversión de modelos DEVS

En este capitulo describimos en detalles los pasos para realizar la transformación, utilizaremos el modelo ya presentado e introduciremos un modelo PowerDEVS con componentes vectoriales con el fin de ilustrar la transformación de estos componentes.

3.1. Modelos DEVS

3.1.1. Archivos PDS

PowerDEVS trabaja principalmente con dos archivos, .PDM y .PDS. Los archivos PDM es utilizado por el editor de modelos, pues contiene información estructural, así como información de posición de los modelos dentro del editor, sus parámetros, lo cual indica nombre, tipo y valor, descripción de los modelos, la cantidad de puertos de cada modelo y las conexiones entre los modelos, los detalles de como esas conexiones son visualizadas por lineas y los detalles del recorrido de esas lineas. Todos estos elementos son utilizados en primera medida con el editor de modelos, pero tambien son utilizados para generar el archivo PDS el cual es el genera el código de la simulación. El archivo PDS contiene información estructural del modelo necesaria para realizar la simulación, en el listado 3 se pueden ver un esbozo de su estructura.

```
Root-Coordinator
 {
  Simulator
   {
    Path = vector\qss_sum_vec.h
    Parameters = "1","-1","1","0","0","0","0","0",3.000000e+00,"N"
   }
   ...
    Coordinator
     {
      ...
     }
   ...
  Simulator
   {
        ...
   }
  EIC
   {
   }
  EOC
   {
   }
  IC
   {
        ...
   }
 }
```

Listado 3: Estructura de un archivo PDS.

Se puede observar un elemento `Root-Coordinator` el cual contiene (marcado entre llaves) una lista de `Simulator` que representan los modelos atómicos y/o `Coordinator` que representan los modelos acoplados y tres listas de conexiones `EIC`, `EOC` y `IC`, conexiones de entrada externa (External Input Connections), conexiones de salida externa (external output connections) y conexiones internas (Internal Connections).

Las lista de conexiones internas (IC) es una lista de par de pares de números naturales, de la forma $(a, b); (c, d)$. Donde el primer par se refiere al origen de la conexión (puerto b de salida) del modelo a y el segundo al fin de la conexión (puerto d de entrada) del modelo c, ambos modelos se refiere a la lista de modelos (atómicos o acoplados) del actual modelo. Es decir el par $(0, 0); (1, 0)$ indica que el puerto 0 del segundo modelo (posición 1) es de entrada y esta conectado al puerto 0 del primer modelo.

Tanto EIC y EOC siguen el mismo patron excepto que replazan el modelo de los puertos de salida y extrada respectivamente por 0. Es decir el elemento $(6,0);(0,1)$ en EOC indica que el puerto 0 del modelo 6 (septima posición) se encuentra conectado con el puerto 1 de salida del modelo acoplado, y un par $(0,0);(2,1)$ en EIC indica que el primer puerto (puerto 0) se encuentra conectado con el modelo 2 (tercera posición) en su puerto 1.

Los modelos acoplados dado que también son modelos DEVS, replican la estructura listado 3.

Para poder leer esta estructura se cuenta con la librería de PowerDEVS[1] la cual nos permite acceder a la estructura desde C++.

3.2. Modelos Atómicos

Internamente los modelos atómicos son identificados por el parámetro `Path` en el archivo PDS, para realizar la tradución de este modelo se utiliza un modelo Modelica el cual debe seguir la siguiente especificación:

- El código debe ser Modelica (μ-modelica) valido y estar ubicado en el mismo directorio (y nombre del archivo) del código C que el modelo atómico PowerDEVS, con el mismo nombre que el archivo .h, pero con extensión .mo, es decir un modelo con `vector\qss_sum_vec.h` utilizara un modelo `vector/qss_sum_vec.h` [2]
- Los parámetros del modelo DEVS deben ser pasado en el parámetro p
- Los valores de entrada del modelo son asociados a la variable u
- Los valores de salida del modelos son asociados a la variable y

Por ejemplo el código del integrador, originalmente ubicado en el archivo qss_integrator.h de PowerDEVS, se ubica en el archivo qss_integrator.mo mostrado en el listado 4 ambos dentro del directorio qss.

```
class QSSIntegrator
  parameter Real p[4]={0,0,0,0,0,0,0,0};
  parameter Real x0 = p[4];
  Real u[1];
  Real y[1](start = {x0});
equation
  der(y[1]) = u[1];
end QSSIntegrator;
```

Listado 4: Modelo qss_integrator.mo

[1] http://sourceforge.net/p/powerdevs/code/HEAD/tree/

[2] El nombre de los archivos se replaza "\" por "/" para permitir algunos modelos cuyos `Path` contiene ese separadores de directorios

El modelo Lotka Volterra cuenta con dos integradores (con los mismo parámetros) representados en el listado 3.2, en este se puede ver el valores de `Path` y `Parameters` que componen todos los modelos atómicos (por supuesto con los valores diferentes).

```
...
Simulator
 {
  Path = qss/qss_integrator.h
  Parameters = "QSS3","1e-6","1e-3","0.5"
 }
 ...
```

Listado 5: Extracto del modelo Lotka Volterra, modelo atómico de un integrator.

Luego de remplazar los parámetros en la variable `p`, se prefijan todas las variables del modelo[3] por el nombre del modelo y su posición en este caso con el prefijo "`QSSIntegrator_1_`", de esta forma podremos combinar varios modelos atómicos de forma que no existan "colisión" de nombres de variables, es decir dos variables de distintos modelos atómicos con el mismo nombre en Modelica.

```
class QSSIntegrator
  parameter Real QSSIntegrator_1_p[4]={0,1e-6, 1e-3, 0.5};
  parameter Real QSSIntegrator_1_x0 = p[4];
  Real QSSIntegrator_1_u[1];
  Real QSSIntegrator_1_y[1](start = {QSSIntegrator_1_x0});
equation
  der(QSSIntegrator_1_y[1]) = QSSIntegrator_1_u[1];
end QSSIntegrator;
```

Listado 6: Transformación parcial de un modelo atómico de un integrator en el modelo de ejemplo Lotka Volterra.

Los parámetros son remplazados en el modelo, evaluándolos en Scilab[4], lo que los transforma en float, los cuales son presentados como reales (Real) en el código.

Los modelos (atómicos) no encontrados son ignorados en la traducción y se reportan en el registro (archivo .log) de la conversión.

Esta transformación se repite para todos los modelos atómicos del archivo .PDS dentro del `Root-Coordinator`, los modelos acoplados (dentro

[3] La única variable que no se prefijada es `time`

[4] Para realizar la evaluación en Scilab se utiliza el mismo mecanismo que provee (y utiliza) PowerDEVS.

de un `Coordinator`) deben ser transformados a un conjunto de atómicos equivalentes. “Plano”

3.3. Modelos Acoplados Planos

Llamamos *Modelos Acoplados Planos* a los Modelos que solo contienen *Modelos Atómicos*. Es decir que no contienen Modelos acoplados.

Cada uno de los modelos atómicos que incluye se transforma de la misma forma que describimos en la sección anterior y dado que no existen “colisiones” de nombres variables, podes combinar las diferentes secciones en un modelo compuesto el cual tendra todas las secciones `equation`, `initial equation` y declaraciones. Luego cada conexión entre Modelos Atómicos es replicada en el código de Modelica resultante. Los modelos Atómicos cuyo entrada (o salida) son escalares son conectados con un ecuación del tipo $u = y$ mientras que los modelos vectoriales son conectados con la misma ecuación, solo que dentro de un `for`.

```
1  model lotka_volterra
2    Real qss_multiplier_0_u[2];
3    Real qss_multiplier_0_y[1];
4    parameter Real QSSIntegrator_1_p[4] = {0,1e-06,0.001,0.5};
5    parameter Real QSSIntegrator_1_x0 = 0.5;
6    Real QSSIntegrator_1_u[1];
7    Real QSSIntegrator_1_y[1](start = {QSSIntegrator_1_x0});
8    parameter Real QSSIntegrator_2_p[4] = {0,1e-06,0.001,0.5};
9    parameter Real QSSIntegrator_2_x0 = 0.5;
10   Real QSSIntegrator_2_u[1];
11   Real QSSIntegrator_2_y[1](start = {QSSIntegrator_2_x0});
12   parameter Real WSum_3_p[9] = {0.1,(-0.1),0,0,0,0,0,0,2};
13   parameter Integer WSum_3_n = integer(2);
14   parameter Real WSum_3_w[WSum_3_n] = WSum_3_p[1:WSum_3_n];
15   Real WSum_3_u[WSum_3_n];
16   Real WSum_3_y[1];
17   parameter Real WSum_4_p[9] = {0.1,(-0.1),0,0,0,0,0,0,2};
18   parameter Integer WSum_4_n = integer(2);
19   parameter Real WSum_4_w[WSum_4_n] = WSum_4_p[1:WSum_4_n];
20   Real WSum_4_u[WSum_4_n];
21   Real WSum_4_y[1];
22   equation
23     qss_multiplier_0_y[1] = qss_multiplier_0_u[1]*qss_multiplier_0_u[2];
24     der(QSSIntegrator_1_y[1]) = QSSIntegrator_1_u[1];
25     der(QSSIntegrator_2_y[1]) = QSSIntegrator_2_u[1];
26     WSum_3_y[1] = WSum_3_u*WSum_3_w;
27     WSum_4_y[1] = WSum_4_u*WSum_4_w;
28     qss_multiplier_0_u[1] = QSSIntegrator_1_y[1];
29     qss_multiplier_0_u[2] = QSSIntegrator_2_y[1];
30     QSSIntegrator_1_u[1] = WSum_3_y[1];
31     WSum_3_u[2] = qss_multiplier_0_y[1];
32     WSum_3_u[1] = QSSIntegrator_1_y[1];
33     QSSIntegrator_2_u[1] = WSum_4_y[1];
34     WSum_4_u[1] = qss_multiplier_0_y[1];
35     WSum_4_u[2] = QSSIntegrator_2_y[1];
36 end lotka_volterra;
```

Listado 7: Modelo Lotka Volterra convertido de PowerDEVS a μ-Modelica

En el listado 7 se puede ver el resultado de la conversión del modelo Lotka Volterra, en el se puede apreciar en las líneas 2 a 21 correspondientes a las declaraciones de las variables de los modelos atómicos y de las líneas 22 a 27 correspondientes a las ecuaciones de estos modelos y de la línea 28 a 35 son las ecuaciones correspondientes a las conexiones entre modelos

atómicos.

En las lineas de conexiones (28 a 35) se puede ver como se utilizan las variables `u` e `y` (prefijadas con sus correspondientes modelos atómicos y posición). Tanto los puertos de entrada, `u`, como los puertos de salida, `y`, son representados en Modelica como arreglos, permitiendo a modelos que cuentan con más de un puerto reflejar este hecho, asociando cada puerto con la posición correspondiente del arreglo. cabe mencionar que existe un desfase, ya que los puerto en PowerDEVS son enumerados desde el cero, mientras que Modelica inician los arreglos de uno, por lo que el puerto n corresponde a la variable $u[n+1]$ si es un puerto de entrada y $y[n+1]$ si es un puerto de salida.

En el ejemplo Lotka Volterra no hay modelos atómicos vectoriales, veremos más adelante un ejemplo, pero es oportuno mencionar que los modelos vectoriales difieren en la forma en que se conectan, dado que la conexión debe realizarse iterando, con un `for`, sobre la primera dimensión del arreglo que representa el puerto, en el caso escalar, que es el caso por omisión, solo alcanza con igualar las variables mencionadas.

3.4. Modelos Acoplados Jerárquicos

En la sección anterior mostramos como son convertidos modelos acoplados planos, para convertir un modelo acoplado jerárquico, es decir un modelos con más modelos acoplados internos, vamos a generar un modelo acoplado plano, equivalente al modelo jerárquico inicial.

Para realizar el aplanado, se recorre recursivamente los modelos acoplados:

- por cada modelo acoplado si solo tiene modelos atómicos, es remplazado por los modelos atómicos internos, los cuales se encuentran conectados sin modificaciones excepto por las conexiones externas, las cuales son reasignadas de forma de mantener las conexiones.
- si el modelo acoplado contiene otros modelos acoplados entonces aplanamos ese modelo recursivamente.

De esta forma obtenemos un modelo con solo modelos atómicos el cual podemos convertir con el procedimiento anteriormente descripto.

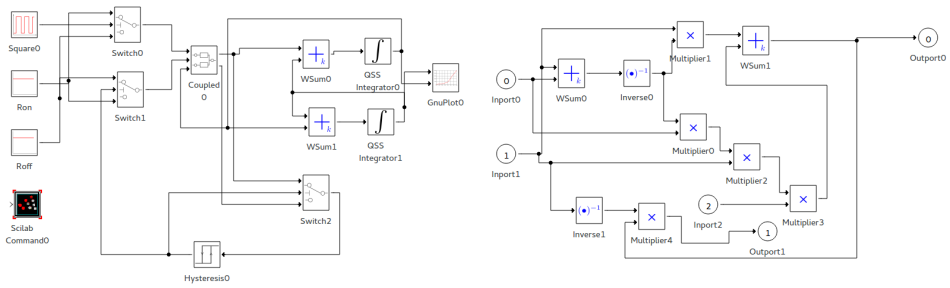


Figura 3.1: Ejemplo de Modelo acoplado(derecha), junto a una detalle del modelo `Coupled0`

En la figura 3.4 se observa a la derecha el modelo de un convertidor Buck (o reductor) es un convertidor de potencia, el cual introduciremos con mayores detalles en una sección posterios, y a la izquierda el modelo acoplado `Coupled0`.

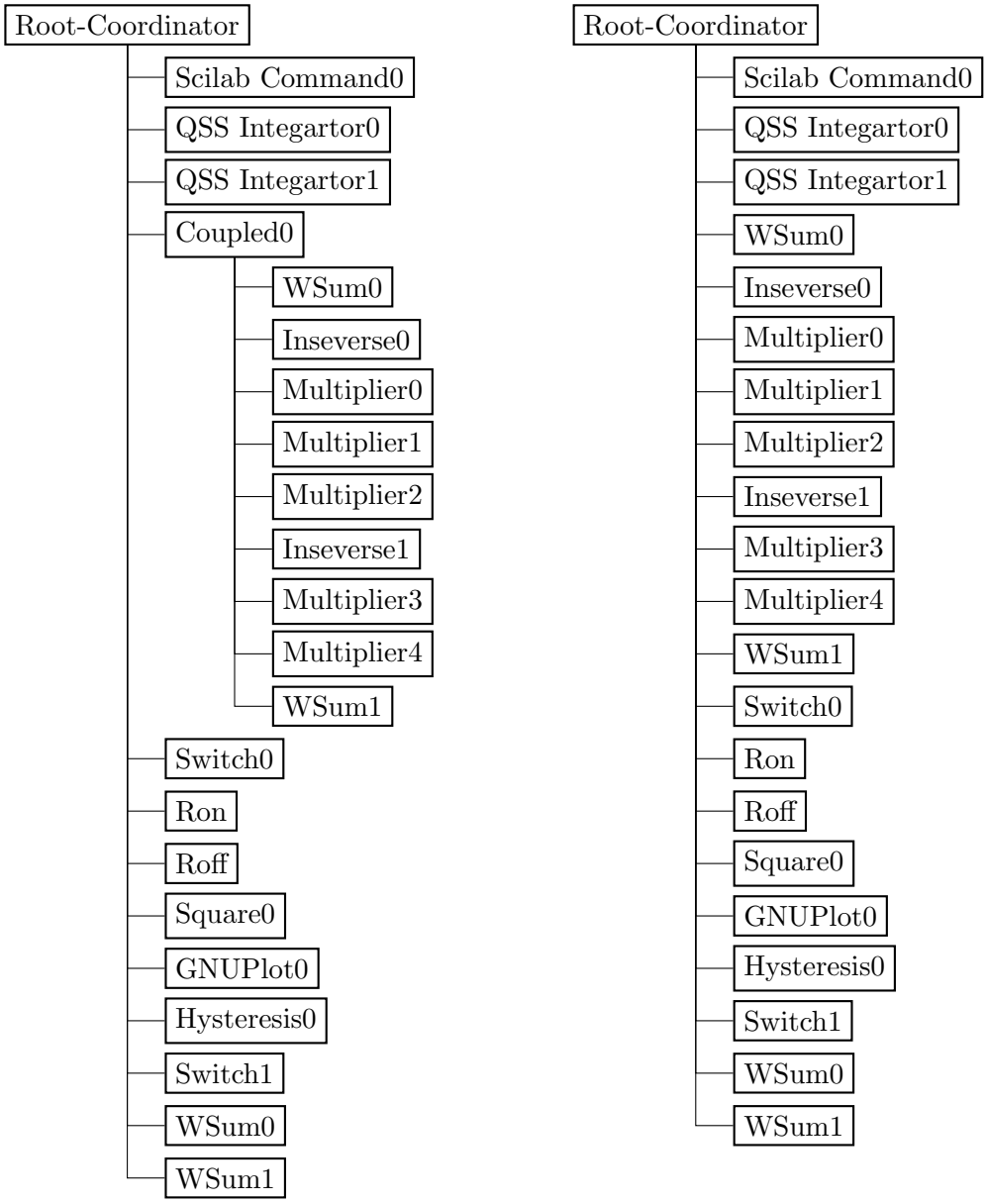


Figura 3.2: Modelo convertidor Buck de potencia, jerarquía original (izquierda) y aplanada (derecha)

En la figura 3.2 se puede observar la forma en que se modifican las jerarquía, esencialmente se mueven los modelos atómicos a la jerarquía superior y se modifican las conexiones de forma que se mantengan los enlaces, es decir todas las conexiones entre modelos atómicos listados despues del modelo acoplado deben ser ajustados para tomar en cuenta los nuevos modelos atómicos agregados.

En el modelos aplanado, se modificaran de listado 3.4 las conexiones de

```
IC
  {
  (13,0);(1,0)
   (12,0);(2,0)
   (8,0);(4,1)
   (4,0);(3,0)
   (5,0);(3,1)
   (3,1);(11,2)
   (11,0);(10,0)
   (2,0);(9,0)
   (2,0);(12,0)
   (2,0);(13,1)
   (3,0);(11,0)
   (3,0);(13,0)
   (10,0);(11,1)
   (10,0);(5,1)
   (1,0);(3,2)
   (1,0);(12,1)
   (1,0);(9,1)
   (6,0);(5,2)
   (6,0);(4,0)
   (7,0);(5,0)
   (7,0);(4,2)
}
```

```
EIC
  {
   (0,2);(4,1)
   (0,0);(2,1)
   (0,0);(0,1)
   (0,1);(3,1)
   (0,1);(5,0)
   (0,1);(7,0)
   (0,1);(0,0)
  }
 EOC
  {
   (6,0);(0,1)
   (8,0);(0,0)
  }
 IC
  {
   (0,0);(1,0)
   (7,0);(8,0)
   (2,0);(3,0)
   (3,0);(4,0)
   (5,0);(6,0)
   (4,0);(8,1)
   (1,0);(2,0)
   (1,0);(7,1)
   (8,0);(6,1)
  }
```

Listado 8: Conexiones del modelo acoplado convertidor Buck de potencia, a la derecha, las conexiones del primera nivel (`Root Coordinator`), a la derecha, las conexiones, external entrada (EIC) y salida (EOC) y las conexiones internas (IC) del modelo acoplado (`Coupled0`).

forma que se respeten las conexiones entre los modelos atómicos, si pasan por un puerto de salida o de entrada, vemos tres tipos de conexiones que deben ser modificadas:

- Conexiones que involucran modelos atómicos ubicados después del modelo acoplado y no están relacionadas con el modelo acoplado, estas conexiones deben ser modificadas dado que insertaremos los modelos atómicos del modelo acoplado que estamos aplanando, ubicados después del modelo acoplado serán desplazados.

```
IC
  {
    (13,0);(1,0)
    (12,0);(2,0)
    (8,0);(4,1)
    (11,0);(10,0)
    (2,0);(9,0)
    (2,0);(12,0)
    (2,0);(13,1)
    (10,0);(11,1)
    (10,0);(5,1)
    (1,0);(12,1)
    (1,0);(9,1)
    (6,0);(5,2)
    (6,0);(4,0)
    (7,0);(5,0)
    (7,0);(4,2)
  }
```

```
IC
  {
    (21,0);(1,0)
    (20,0);(2,0)
    (16,0);(12,1)
    (19,0);(18,0)
    (2,0);(17,0)
    (2,0);(20,0)
    (2,0);(21,1)
    (18,0);(19,1)
    (18,0);(13,1)
    (1,0);(20,1)
    (1,0);(17,1)
    (14,0);(13,2)
    (14,0);(12,0)
    (15,0);(13,0)
    (15,0);(12,2)
  }
```

Listado 9: Conexiones internas, como se encontraban originalmente a la izquierda y modificadas a la derecha

En el listado 3.4 se puede ver a la derecha las conexiones que fueron afectadas y a la izquierda como fueron afectadas, en nuestro ejemplo se sumo 8 (ya que insertaremos esa cantidad de modelos) a los modelos mayores que 3 (ya que el modelos acoplado que estamos aplanando se encuentra en es posición).

- Agregamos las conexiones internas del modelo acoplado que eliminaremos al modelo acoplado `Root-Coordinator`, estas conexiones deben ser modificadas ya que los modelos atómicos insertados son insertados en la posición del modelo acoplado eliminado.

  En el listado 3.4 se pueden ver las conexiones como se encontraban originalmente (izquierda) y como serán insertadas (derecha) estas son el resultados de desplazar los modelos en nuestro ejemplo 3 posiciones por lo que sumamos ese desplazamiento a los modelos.

  Se puede ver en la figura 3.4 un esquema de conexiones, donde el modelo $S2$ es un modelo acoplado que contiene el modelo $S3$. El modelo atómico $S1$ se conecta a travéz del puerto $p2$. Entonces si consideramos que $p2$ es un puerto de entrada externa, veremos las siguientes conexiones:

  - IC : `(S1,p1);(S2,p2)`
  - EIC : `(0,p2);(S3,p3)`

```
IC
 {
  (0,0);(1,0)
  (7,0);(8,0)
  (2,0);(3,0)
  (3,0);(4,0)
  (5,0);(6,0)
  (4,0);(8,1)
  (1,0);(2,0)
  (1,0);(7,1)
  (8,0);(6,1)
 }
```

```
IC
 {
  (3,0);(4,0)
  (10,0);(11,0)
  (5,0);(6,0)
  (6,0);(7,0)
  (8,0);(9,0)
  (7,0);(11,1)
  (4,0);(5,0)
  (4,0);(10,1)
  (11,0);(9,1)
 }
```

Listado 10: Conexiones internas del modelo acoplado a eliminar, a la izquierda como aparecen originalmente, a la derecha como serán insertados

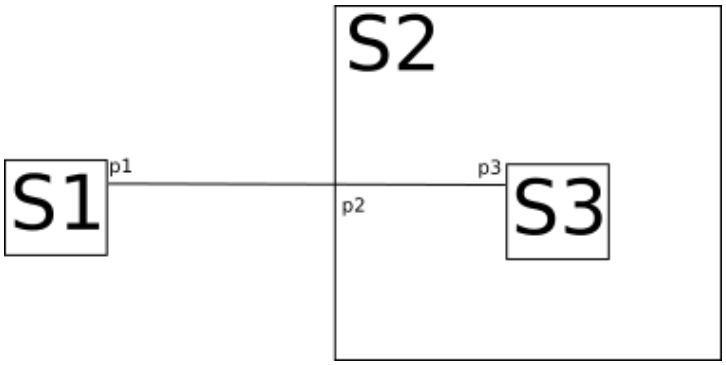


Figura 3.3: Esquema de conexiones, el modelo `S1` y `S3` son modelo atómico, `S2` es el modelo acoplado que estamo aplanando. `S1` y `S2` estan conectados a travéz de los puertos `p1` y `p2` respectivamente. Luego, el modelo `S3` conecta con el exterior a travéz del modelo acoplado que lo contiene por el puerto `p2` y wl puerto `p3`.

entonces la conexiones en el modelo aplanado es `(`$S1$`,`$p1$`);(`$S3$`,`$p3$`)` y $S1$ debe ser desplazado segun la cantidad de modelos que contiene $S2$ y $S3$ debera ser desplazado segun la posición de $S2$.

Si consideramos que $p2$ es un puerto de salida las conexiones seran:

- IC : `(S2,p2);(S1,p1)`
- EOC : `(S3,p3);(0,p2)`

entonces la conexiones en el modelo aplanado es `(`$S3$`,`$p3$`);(`$S1$`,`$p1$`)` e igual que en el caso anterior, $S1$ debe ser desplazado segun la cantidad de modelos que contiene $S2$ y $S3$ debera ser desplazado segun la posición de $S2$.

```
IC                EOC               IC
 {                 {                 {
  (3,1);(11,2)      (6,0);(0,1)       (9,0);(19,2)
  (3,0);(11,0)      (8,0);(0,0)       (11,0);(19,0)
  (3,0);(13,0)     }                  (11,0);(21,0)
 }                                    }
```

Listado 11: Conexiones internas desde el modelo acoplado hacia otro modelo (izquierda), conexiones externas de salida (centro), conexiones internas a agregar al modelo aplanando(derecha).

```
IC                EIC               IC
 {                 {                 {
  (4,0);(3,0)       (0,2);(4,1)       (12,0);(5,1)
  (5,0);(3,1)       (0,0);(2,1)       (12,0);(3,1)
  (1,0);(3,2)       (0,0);(0,1)       (13,0);(6,1)
 }                  (0,1);(3,1)       (13,0);(8,0)
                    (0,1);(5,0)       (13,0);(10,0)
                    (0,1);(7,0)       (13,0);(3,0)
                    (0,1);(0,0)       (1,0);(7,1)
                   }                 }
```

Listado 12: Conexiones internas hacia el modelo acoplado (izquierda), conexiones externas de entrada(centro), conexiones internas a agregar al modelo aplanando(derecha).

Agregando estas conexiones se elimina el modelo acoplado y se preserva las conexiones, y recorriendo el árbol en profundidad, aplanando los modelos acoplado más profundos primero podemos saber que solo estamos aplanando el modelo más profundo unicamente.

3.5. Modelos Vectoriales

Como ya mencionamos, los modelos vectoriales son modelos atómicos en el cual las entradas y/o salidas son vectoriales. En los modelos Modelica, las variables `u` e texttty representan los puertos de entrada y salida respectivamente, por lo que serán del tipo arreglo de arreglos de reales, mientras que en los modelos atómicos no vectoriales (escalares) tienen variables que representan los puertos de entrada y salida como arreglos de reales.

En el momento cuando las estructura del modelo acoplado es transformada a Modelica, es decir cuando se agregan las ecuaciones de igualdad entre las variables de entrada y salida de diferentes modelos, según lo dispongan la estructura del modelo, es decir las conexiones internas, en PowerDEVS no existe una forma de determinar si un modelo es vectorial o no, por lo que debemos indicarlo dentro del modelo Modelica, ya que si no lo hace-

mos agregaremos una ecuación que iguala un arreglo con una entrada de un arreglo, para prevenir este problema los modelos vectoriales deben indicarse con la anotación de Modelica $PD2MO$, por ejemplo:

- `annotation(PD2MO = {Scalar, Vector});` entrada escalar y salida vectorial
- `annotation(PD2MO = {Vector, Scalar});` entrada escalar y salida vectorial
- `annotation(PD2MO = {Vector, Vector});` entrada y salida vectoriales.
- `annotation(PD2MO = {Scalar, Scalar});` entrada y salida son escalares, este es el caso por omisión y no es necesario declararlo.

De esta forma podemos realizar la conexiones correctamente y generar un error en caso de encontrar una conexión escalar con una vectorial.

3.6. Transformaciones Extras

Existen dos tipos de transformaciones extras que deben ser aplicadas con el fin de que el código generado sea μ-Modelica, en particular estas transformaciones se agrupan en dos:

- Causalización de variables, este es un programa descripto en [**Mod15**]
- Transformaciones de Modelica a μ-Modelica, si bien existe un programa con esta finalidad este no implementa (al momento de escribir este texto) las transformaciones que necesitamos por lo que son descriptas en la sección 4.4.

Capítulo 4

Detalles de la implementación

A continuación se presenta el pseudo código que implementa las transformaciones descriptas en este trabajo, el código completo puede encontrarse en `https://github.com/lucciano/pd2mo`, el cual utiliza dos librerias, Modelica C Compiler [1] el cual nos permite manipular la estructura de los modelos y evaluar los parámetros. y libreria de PowerDEVS [2] para leer los archivos PDS.

El programa esta separado en 4 módulos:

4.1. Programa Principal

El Programa principal en el archivo main.cpp, el cual es responsable de la interfaz con el usuario (linea de comando) y lanzar la transformación de la simulación, asi como establecer los archivos desde donde se lee y hacia donde se escriben la simulación de powerDEVS y Modelica, respectivamente.

Algorithm 1 main(src_infile)

```
1: modelCoupled *cm ← parsePDS(QString::fromStdString(src_infile));
2: modelCoupled *qm ← flatter::flat(cm);
3: Pd2Mo q ← Pd2Mo();
4: q.transform(flatted, modelname, &outfile, &oFlogfile);
5: AST_StoredDefinition sd ← parseFile(src_outfile.c_str(),&amp;amp;r);
6: mda *m ← new mda();
7: If *i ← new If();
8: outfile ≪ m→visitClass(prod→visitClass( i→visitClass(
   *sd→models()→begin()))) ≪ endl;
```

4.2. Transformación Principal

La clase *Pd2Mo* implementa las principales partes de la transformación, la cual incluye abrir el archivo PDS, e invocar el aplanado, obtener los diferentes modelos modelica que representan los modelos atómico, prevenir la colisión de nombres, crear el modelo final y realizar las conexiones.

[1] http://sourceforge.net/projects/modelicacc/
[2] http://sourceforge.net/projects/powerdevs/

Algorithm 2 Pd2Mo::transform()

1: modelCoupled *model ← parsePDS(qfilename);
2: AST_ClassList classList ← getAsClassList(model);
3: int j ← 0
4: **for** class en classList **do**
5: **if** La clase esta traducida a μModelica **then**
6: Prefijamos las variables con el nombre del modelo *class* y la posición j que ocupan en la lista;
7: Remplazamos la entrada *class* dentro de la lista por su copia producida en el paso anterior;
8: **end if**
9: **end for**
10: Creamos un modelo $modeloMo$;
11: **for** class en classList **do**
12: Combinamos el modelo *class* con el $modeloMo$;
13: **end for**
14: **for** ic en Conexión Interna del Modelo **do**
15: Las conexiones ic estan definidas como dos pares de números, cada par señalan número de modelo y número de puerto, en este caso los puertos deben ser desfasados en uno, pues los puertos en nuestra representación son los sub-indices de u e y para cada modelo, pero el primer elemento de los arreglos en Modelica comienzan.
16: **if** los modelos de ic son Escalares **then**
17: Se agrega la ecuación que representa la conexión entre los modelos;
18: **else if** los modelos de ic son Vectoriales **then**
19: Se agregan N ecuaciones indexadas por i que representa la conexión, vectorial entre los modelos mediante una sentencia `For`;
20: **else**
21: No se conoce la conexión;
22: **end if**
23: **end for**

4.3. Aplanado de modelos acoplados

La clase *flatter* implementa el aplanado de los modelos acoplados descripto en la sección 3.4.

Algorithm 3 flatter::flat

```
1: for ModeloHijo en Lista de Modelos do
2:     if Tipo de ModeloHijo es COUPLED then
3:         for ModeloHijo2 en Lista de ModeloHijo→ModeloHijo do
4:             if Tipo de ModeloHijo2 es ATOMIC then
5:                 Copiamos el ModeloHijo2 al ModeloResultado;
6:             else
7:                 Copiamos el aplanado de ModeloHijo2;
8:             end if
9:             for Conexión del Modelo do
10:                if Si la conexión involucra un modelo "aun no procesado" then
11:                    Las conexiones deben ser modificadas teniendo en cuenta los modelos agregados en el aplanado;
12:                end if
13:                if Si la conexión involucra como destino el modelo acoplado ModeloHijo then
14:                    Se crea una nueva conexión (en ModeloResultado) entre los modelos agregado recientemente según la conexión del puerto de entrada del ModeloHijo y el origen de la conexión;
15:                    La conexión se marca para ser borrada;
16:                end if
17:                if Si la conexión involucra como origen el modelo acoplado ModeloHijo then
18:                    Se crea una nueva conexión (en ModeloResultado) entre los modelos agregado recientemente según la conexión del puerto de salida del ModeloHijo y el destino de la conexión;
19:                    La conexión se marca para ser borrada;
20:                end if
21:                if Si la conexión fue marcada then
22:                    Se borra la conexión
23:                end if
24:            end for
25:        end for
26:    else
27:        Copiamos el nodo ModeloHijo al ModeloResultado
28:        Copiamos las conexiones del ModeloHijo y cualquier otro ModeloHijo que ya haya sido procesado
29:    end if
30: end forreturn ModeloResultado
```

4.4. Transformaciones para μ-Modelica

Tanto la clase `mda`, `prodint` y `If` son implementadas con el patron de diseño de visitadores sobre el árbol sintáctico abstracto[3], por lo que cada clase es implementada heredando de una clase común (Traverser), la cual retorna una copia del AST y remplaza una parte este según sea el objetivo de la clase.

Estas transformaciones son necesarias dado que al momento de realizar este trabajo no son soportadas por el QSS-Solver, ya que no son μ-Modelica valido. Arreglos multidimensionales no están soportados dado que no es μ-Modelica valido, ya que los arreglos son de una sola dimensión, mientras que el producto vectorial de dos vectores no estaba implementado en un principio, ya se encuentra implementado en una versión de desarrollo, y la transformación `If` fue implementada para simplificar le código final.

- `mda`: Remplaza expresiones de la forma `X[N,k]`, donde $k \in \mathbb{N}$ o evalúa a una variable que evalúa a una expresión $\in \mathbb{N}$, es remplazado por `X_k[N]`.

```
Real IndexShift_2_u[IndexShift_2_N,1];
                    ⇓
Real IndexShift_2_u_1[IndexShift_2_N];
```

- *prodint*: Remplaza expresiones de la forma $u[i, 1 : nin] * w$ por expresiones de la forma u[i,1] * w[1] + u[i,2] * w[2] .... + u[i,nin] * w[nin], donde $nin \in \mathbb{N}$ o evaluá a una variable que evaluá a una expresión $\in \mathbb{N}$

```
VectorSum_3_y_1[VectorSum_3_i] =
    VectorSum_3_u[VectorSum_3_i, 1:VectorSum_3_nin] * VectorSum_3_w;
```

(Donde `VectorSum_3_nin = 4` y `VectorSum_3_w` tiene dimensión 4, lo que es necesario para que quede definida el producto de dos vectores en modelica.)

⇓

```
VectorSum_3_y[1,VectorSum_3_i] =
    VectorSum_3_u[1,VectorSum_3_i]*VectorSum_3_w[1]+
    VectorSum_3_u[2,VectorSum_3_i]*VectorSum_3_w[2]+
    VectorSum_3_u[3,VectorSum_3_i]*VectorSum_3_w[3]+
    VectorSum_3_u[4,VectorSum_3_i]*VectorSum_3_w[4];
```

[3]un árbol de sintaxis abstracta (AST), o simplemente un árbol de sintaxis, es una representación de árbol de la estructura sintáctica abstracta (simplificada) del código fuente escrito en cierto lenguaje de programación.

- If: Remplaza expresiones de la forma $if(v)eq_1elseeq_2$ si v evaluá a un booleano (a partir de parámetros o constantes, es decir en análisis estático) se remplaza por eq_1 o eq_2 si es v es verdadero o falso respectivamente.

```
if IndexShift_2_Shift > 0 then
  for IndexShift_2_i in 1:IndexShift_2_N-IndexShift_2_Shift loop
    IndexShift_2_y_1[IndexShift_2_i+IndexShift_2_Shift] =
      IndexShift_2_u_1[IndexShift_2_i];
  end for;
  for IndexShift_2_i in 1:IndexShift_2_Shift loop
    IndexShift_2_y_1[IndexShift_2_i] = 0;
  end for;
else
  for IndexShift_2_i in 1:IndexShift_2_N-IndexShift_2_Shift loop
    IndexShift_2_y_1[IndexShift_2_i] =
              IndexShift_2_u_1[IndexShift_2_i - +IndexShift_2_Shift];
  end for;
  for IndexShift_2_i in IndexShift_2_N + IndexShift_2_Shift : IndexShift_2_N loop
    IndexShift_2_y_1[IndexShift_2_i] = 0;
  end for;
end if;
```

(Con `Shift` > 0)

$\Downarrow$

```
for IndexShift_2_i in 1:IndexShift_2_N-IndexShift_2_Shift loop
  IndexShift_2_y_1[IndexShift_2_i+IndexShift_2_Shift] =
      IndexShift_2_u_1[IndexShift_2_i];
end for;
for IndexShift_2_i in 1:IndexShift_2_Shift loop
  IndexShift_2_y_1[IndexShift_2_i] = 0;
end for;
```

Capítulo 5

Ejemplos y Resultados

En este capitulo mostramos los resultados del presente trabajo, comparamos los resultados de la ejecución de 5 modelos, los tiempos de ejecución y comparamos los resultados obtenidos. Los modelos ejecutados tanto los originales, en PowerDEVS, y los modelos transformados en μ-Modelilca se encuentran en `https://github.com/lucciano/pd2mo/tree/master/doc/tesina/src`

5.1. Comparación de performance

A continuación por cada uno de los modelos se muestra su modelo en Powerdevs seguido de una gráfica de valores en tiempo de las simulaciones, a la izquierda se muestra el resultado en PowerDEVS y a la derecha los de QSS-Solver convertidos por la herramienta desarrollada.

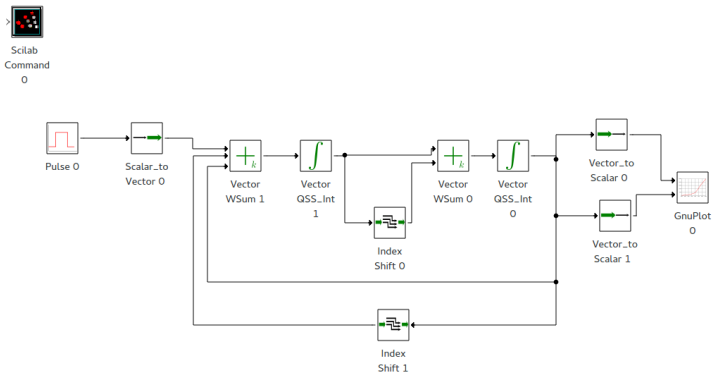
5.2. Líneas de Transmisión

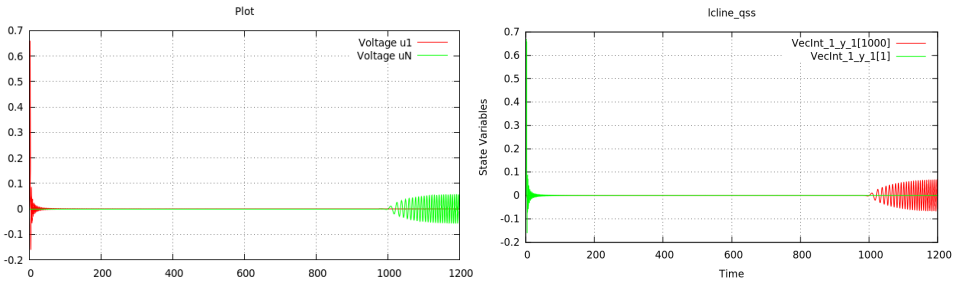
El siguiente sistema de ecuaciones representan un modelo a parámetros concentrados de una línea de transmisión formada por N secciones de circuitos LC:

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

para $j = 1 \ldots N$
Consideramos también un pulso de entrada:

$$v_0(t) = \begin{cases} 1 \text{ si } t < 1 \\ 0 \text{ en caso contrario} \end{cases} \tag{5.1}$$

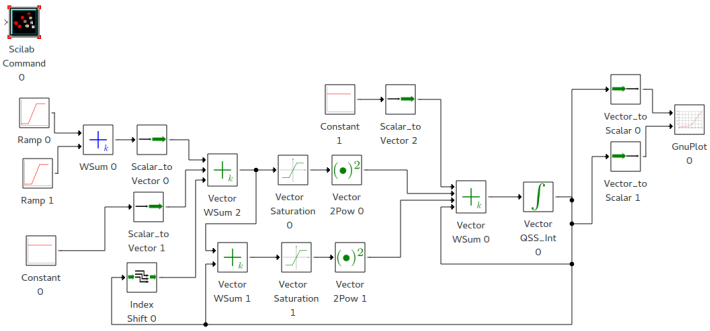


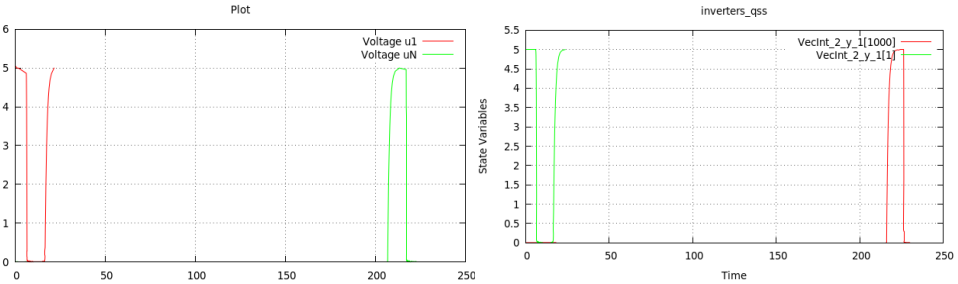


5.3. Inversores Lógicos

El siguiente modelo representa una cadena de m inversores lógicos,

$$\dot{\omega}_1 = U_{op} - \omega_1(t) - \Upsilon g(u_{in}(t), \omega_1(t))$$
$$\dot{\omega}_j = U_{op} - \omega_j(t) - \Upsilon g(\omega_{j-1}(t), \omega_j(t)) \text{ donde } j = 2, 3, .., m$$





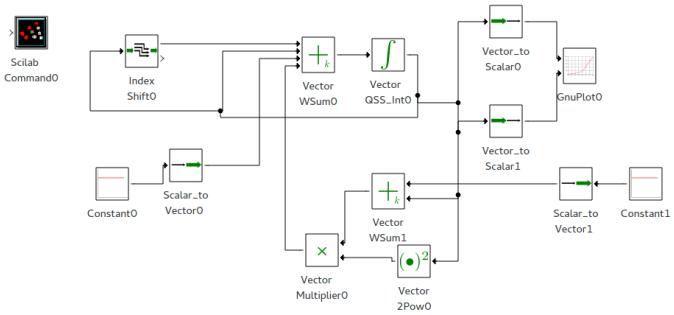
5.4. Advection-diffusion-reaction

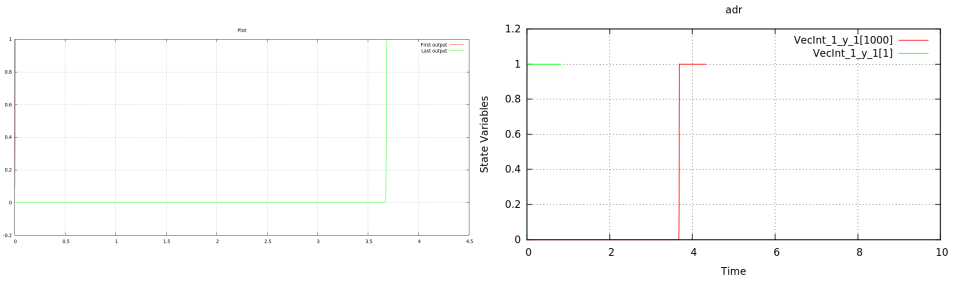
El modelo de equaciones Advection-diffusion-reaction (ADR) provee las bases para describir fenomenos de tranferencias de calor y masa, donde la cantidad de interes $u(x,t)$ puede ser temeratura en la conducción de calor o concentración de una sustancia química.

La ecuación

$$\frac{du(x,t)}{dt} + a\frac{du(x,t)}{dx} = d\frac{d^2u(x,t}{d^2x} + r(u(x,t)^2 - u(x,t)^3)$$

corresponde al modelo ADR, donde a,d y r son parametros expresando coeficientes de advección, difusión y reactión





5.5. Convertidor de Voltaje

El siguiente modelo es un tipo de convertidor DC - DC que obtiene a su salida un voltaje continuo menor que a su entrada, manteniendo una una alta eficiencia (superior al 95 % con circuitos integrados) y autoregulación.

$$\frac{di_L}{dt} = \frac{-u_C - R_D i_D}{L}$$
$$\frac{du_C}{dt} = i_D\frac{i_D}{C} - \frac{u_C}{R_L C}$$

donde

$$i_D = \frac{R_s i_L - u_C - U}{R_D + R_s}$$

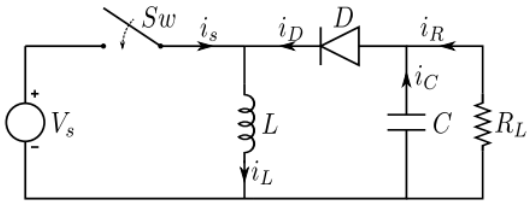


Figura 5.1: Esquema electrico de convertidor de voltaje

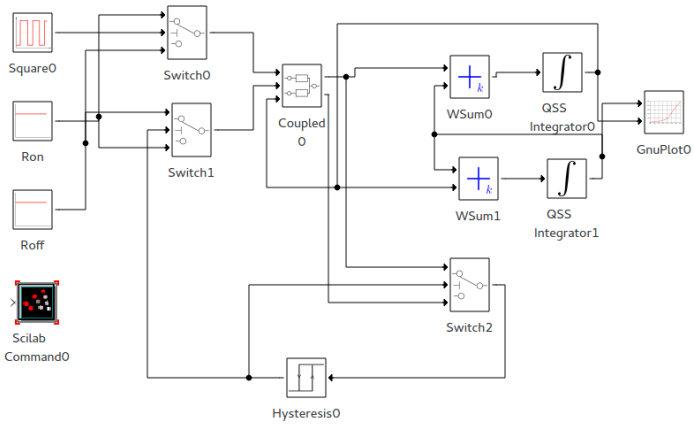


Figura 5.2: Modelo Convertidor de voltaje

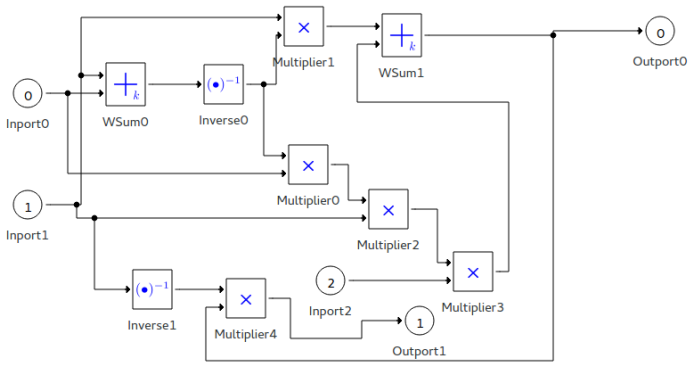
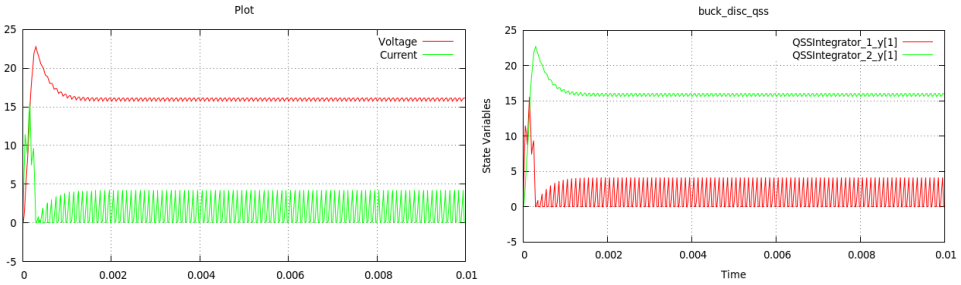


Figura 5.3: Coupled0 (Incluido en Convertidor de voltaje)



5.6. Ecuaciones Lotka-Volter

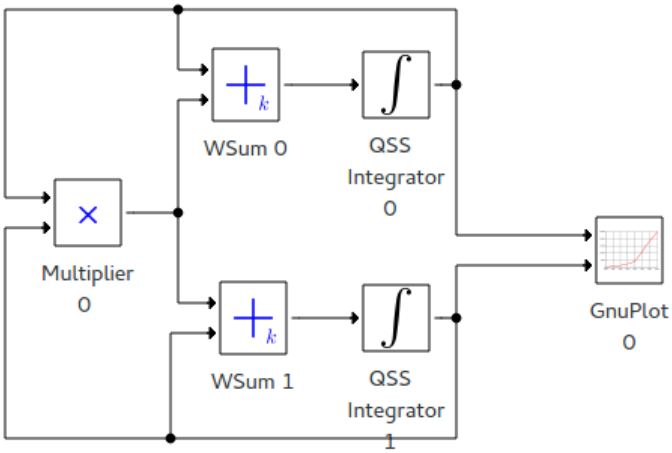
El sistema de ecuaciones Lotka-Volter, es un sistema de ecuaciones diferenciales de primer orden, no lineales, utilizadas para describir dinámicas de sistemas biológicos en el cual dos especies interactúan, una como presa y otra como depredador, se definen como:

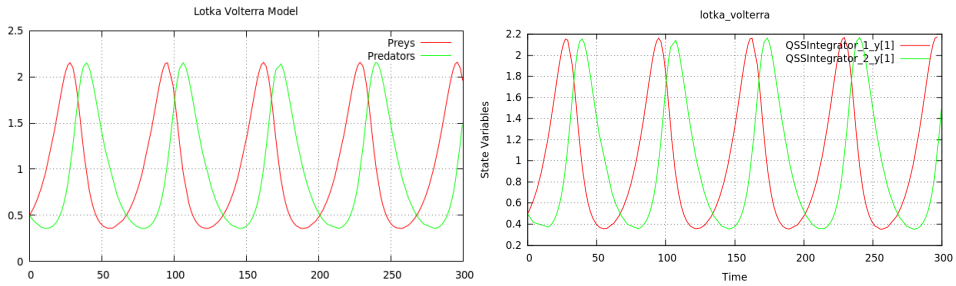
$$\frac{dx}{dt} = x(\alpha - \beta y)$$
$$\frac{dy}{dt} = y(\gamma - \delta x)$$

donde:

- y es el número de algún predador (por ejemplo, un lobo);
- x es el número de sus presas (por ejemplo, conejos);
- t representa el tiempo; y
- α, β, γ, δ son parámetros que representan las interacciones de las dos especies.

Este sistema es representado por el modelo siguiente:





5.7. Resultados

Las pruebas fueron realizadas en una PC Intel® Core™ i7-3632QM CPU @ 2.20GHz con 16 GB de memoria RAM. Los tiempos observados no deben ser considerados como absolutos ya que variaran de un sistema a otro, pero las mejoras relativas en los tiempos de ejecución deberían mantenerse.

Los resultados obtenidos son los tiempos observados luego de ejecutar la simulación en PowerDEVS (P.DEVS) y QSS-Solver (QSS-S), ambas mediciones son reportadas por los motores de simulación, en milisegundos (ms), todas las simulaciones utilizan el método de integración QSS3 excepto los que se especifica LIQSS2.

Modelos	**P.DEVS(ms)**	**QSS-S. (ms)**	**Mejora (%)**
Lineas de Transmisión 1200s, N=1000	76402	34982.5	54
Inversores(LIQSS2) 250s, N=1000	25046	7694.44	69
ADR(LIQSS2) 10s, N=1000	6089	568.772	90
Convertidor de Voltaje 0.01s,	268	10.3802	96
Lotka Voltera 300s	11	2.75132	81

En la tabla se puede ver una mejora entre 54 y 96 % entre todos los modelos, y entre 54 y 90 % para los modelos vectoriales que son nuestro principal objetivo, ya que son la forma más simple de realizar grandes modelos, que es lo que esperaba dado que al cambiar de formalismo se evita todo los mensajes entre modelos caracteristico del formalismo DEVS, además de aprobechar la velocidad del QSS-Solver.

Capítulo 6

Conclusiones y Trabajos futuros

6.1. Conclusiones

En el presente trabajo se mostró, en detalles, la implementación de una herramienta capas de convertir modelos de PowerDEVS en modelos μ-Modelicar. La cual nos permite un ahorro en el tiempo de simulación cercana al 90 % en modelos grandes, mostramos además como se mantiene los valores de la simulación a través de comparar las gráficas de los resultados obtenidos.

6.2. Trabajos Futuros

Para poder convertir una mayor variedad de modelos, en PowerDEVS, es necesario escribir más modelos en modelica. Para convertir modelos de tiempo real sera necesario expandir el QSS-Solver para soportarlos. Para facilitar la integración se planeo integrar el conversor de forma que pueda ser ejecutado desde la interfaz gráfica de PowerDEVS, posiblemente con la opción de ejecutarlo en el QSS SOlver. Durante el desarrollo de este trabajo, la libreria modelicacc [1], ha sido actualizada con un nuevo parser, el cual deberá ser actualizado en este trabajo.

La expansión de variables vectoriales, donde los valores provienen de expresiones del entorno scilab, es siempre tratado como un valor escalar, y no como uno vectorial, para determinar como se lo debería tratar es necesario realizar un analisis de los tipos de las variables expandidas. Lo cual podria mejorar los mensajes de error cuando se genera el modelo final.

[1] http://sourceforge.net/projects/modelicacc/?source=directory

Apéndice A

Modelos Creados

A continuación mostramos los modelos μModelica, utilizados para realizar la comparación de performance:

```
class Constant
  parameter Real p[1]={1};
  parameter Real k = p[1];
  Real y[1];
equation
  y[1] = k;
end Constant;
```

Listado 13: data/sources/constant_sci.mo

```
model pulse_sci
  constant Real p[4] = {1, 1, 1, 2};
  parameter Real low = p[1];
  parameter Real amplitude = p[2];
  parameter Real ti = p[3];
  parameter Real tf = p[4];
  discrete Real d(start = low);
  Real y[1];
equation
  y[1] = pre(d);
algorithm
  when time > ti then
    d := low + amplitude;
  end when;
  when time > tf then
    d := low;
  end when;
end pulse_sci;
```

Listado 14: data/sources/pulse_sci.mo

```
model ramp_sci
  constant Real p[5] = {5, 5, 5, 1, 2};
  parameter Real t0 = p[1];
  parameter Real tr = p[2];
  parameter Real v = p[3];
  discrete Real s, e;
  Real y[1];
initial algorithm
  e := 0;
  s := 0;
equation
  y[1] = (1-pre(e))*pre(s) * ((time-t0) * v / tr)+ pre(e)*v;
algorithm
  when time > t0 then
    s := 1;
  end when;
  when time > t0 + tr then
    e := 1;
  end when;
end ramp_sci;
```

Listado 15: data/sources/ramp_sci.mo

```
model square_sci
  constant Real p[3] = {1, 440, 75};

  parameter Real amplitude = p[1];
  parameter Real freq = p[2];
  parameter Real DC = p[3]/100;
  discrete Real lev(start = 1);
  discrete Real next(start = 0);
  Real y[1];
initial algorithm
  next:= DC/freq;
equation
  y[1] = pre(lev)*amplitude;
algorithm
  when time > next then
        lev:=1-lev;
        next:=time+lev*DC/freq+(1-lev)*(1-DC)/freq;
  end when;
end square_sci;
```

Listado 16: data/sources/square_sci.mo

```
model hysteretic
  constant Real p[4] = {1, 2, 3, 4};
  parameter Real xl = p[1];
  parameter Real xu = p[2];
  parameter Real yl = p[3];
  parameter Real yu = p[4];
  Real u[1];
  Real y[1];
  discrete Real state;
equation
  y[1] = state;
algorithm
  when time > 0  then
    if u[1] > xu then
      state := yu;
    end if;
    if u[1] < xl then
      state := yl;
    end if;
  end when;
  when u[1] > xu then
    state := yu;
  end when;
  when u[1] < xl then
    state := yl;
  end when;
end hysteretic;
```

Listado 17: data/qss/hysteretic.mo

```
class inverse_function
  Real u[1];
  Real y[1];
equation
  y[1] = 1/u[1];
end inverse_function;
```

Listado 18: data/qss/inverse_function.mo

```
class QSSIntegrator
        parameter Real p[4]={0,0,0,0,0,0,0,0};
  parameter Real x0 = p[4];
  Real u[1];
  Real y[1](start = {x0});
equation
  der(y[1]) = u[1];
end QSSIntegrator;
```

Listado 19: data/qss/qss_integrator.mo

```
class qss_multiplier
  Real u[2];
  Real y[1];
equation
  y[1] = u[1] * u[2];
end qss_multiplier;
```

Listado 20: data/qss/qss_multiplier.mo

```
class qss_quantizer
  parameter Real p[2]={1,0};
  parameter Real dQ = p[1];
  discrete Real level(start=0);
  Real u[1];
  Real y[1];
equation
  y[1]=pre(level);
initial algorithm
  if u[1]>level+dQ then
    level:=level+dQ;
  end if ;
  if u[1]<level then
    level:=level-dQ;
  end if ;
algorithm
  when u[1]>level+dQ then
    level:=level+dQ;
  end when;
  when u[1]<level then
    level:=level-dQ;
  end when;

end qss_quantizer;
```

Listado 21: data/qss/qss_quantizer.mo

```
model qss_switch
  parameter Real p[1] = {0};
  parameter Real level = p[1];
  Real u[3];
  Real y[1];
  discrete Real d;
equation
  y[1] = u[1] * d + u[3] * (1 - d);
initial algorithm
  if u[2] > level then
    d := 1;
  elseif u[2] < level then
    d := 0;
  end if;
algorithm
  when u[2] > level then
    d := 1;
  elsewhen u[2] < level then
    d := 0;
  end when;
end qss_switch;
```

Listado 22: data/qss/qss_switch.mo

```
class WSum
        parameter Real p[9]={0,0,0,0,0,0,0,0,0};
        parameter Integer n= integer(p[9]);
        parameter Real w[n] = p[1:n];
         Real u[n];
        Real y[1];
equation
        y[1]=u*w;
end WSum;
```

Listado 23: data/qss/qss_wsum.mo

```
class IndexSelector
  parameter Real p[3] = {3, 9, 100};
  parameter Integer L = p[1];
  parameter Integer H = p[2];
  constant Integer N = p[3];
  Real u[N, 1];
  Real y[N, 1];
equation
  for i in 1:N loop
    y[i, 1] = if i > L and i <= H then u[i, 1] else 0;
  end for;
  annotation(PD2MO = {Vector, Vector});
end IndexSelector;
```

Listado 24: data/vector/IndexSelector.mo

```
class Vec2Scalar
        parameter Real p[2]={0, 40};
        parameter Integer Index=p[1] + 1;
        constant Integer N = p[2];
        Real u[N,1];
        Real y[1];
equation
        y[1]=u[Index,1];
        annotation (PD2MO={Vector,Scalar});
end Vec2Scalar;
```

Listado 25: data/vector/vec2scalar.mo

```
class Scalar2Vector
  constant Real p[2] = {-1, 10};
  constant Integer N = integer(p[2]);
  constant Integer Index = integer(p[1]);
  Real u[1];
  Real y[N, 1];
equation
  if Index == (-1) then
    for i in 1:N loop
      y[i, 1] = u[1];
    end for;
  else
    for i in 1:Index loop
      y[i, 1] = 0;
    end for;
    y[Index + 1, 1] = u[1];
    for i in Index + 2:N loop
      y[i, 1] = 0;
    end for;
  end if;
  annotation(PD2MO = {Scalar, Vector});
end Scalar2Vector;
```

Listado 26: data/vector/scalar2vec.mo

```
class vector_sum
  parameter Real p[2] = {1, 10};
  constant Integer N = p[2];
  Real u[N, 1];
  Real y[1];
  parameter Real K = p[1];
equation
  y[1] = K * sum(u[:,1]);
  annotation(PD2MO = {Vector, Scalar});
end vector_sum;
```

Listado 27: data/vector/vector_sum.mo

```
class VecInt
  parameter Real p[5] = {0, 10, 0, 0, 10};
  constant Integer N = p[5];
  parameter Real x0 = p[4];
  Real u[N, 1];
  Real y[N, 1];
initial algorithm
  for i in 1:N loop
    y[i, 1] := x0;
  end for;
equation
  for i in 1:N loop
    der(y[i, 1]) = u[i, 1];
  end for;
  annotation(PD2MO = {Vector, Vector});
end VecInt;
```

Listado 28: data/vector/qss_integrator_vec.mo

```
class IndexShift
  constant Real p[2] = {-1, 10};
  constant Integer Shift = integer(p[1]);
  constant Integer N = integer(p[2]);
  Real u[N, 1];
  Real y[N, 1];
equation
  if Shift > 0 then
    for i in 1:N - Shift loop
      y[i + Shift, 1] = u[i, 1];
    end for;
    for i in 1:Shift loop
      y[i, 1] = 0;
    end for;
  else
    for i in 1:N + Shift loop
      y[i, 1] = u[i - Shift, 1];
    end for;
    for i in N + Shift + 1:N loop
      y[i, 1] = 0;
    end for;
  end if;
  annotation(PD2MO = {Vector, Vector});
end IndexShift;
```

Listado 29: data/vector/index_shift.mo

```
class VectorSum
  parameter Real p[10] = {1, 1, 0, 0, 0, 0, 0, 0, 2, 1000};
  constant Integer N = p[10];
  constant Integer nin = p[9];
  parameter Real w[nin] = p[1:nin];
  Real u[N, nin];
  Real y[N, 1];
equation
  for i in 1:N loop
    y[i, 1] = u[i, 1:nin] * w;
  end for;
  annotation(PD2MO = {Vector, Vector});
end VectorSum;
```

Listado 30: data/vector/qss_sum_vec.mo

```
class qss_multiplier_vec
  parameter Real p[2] = {1};
  constant Integer N = p[1];
  Real u[N, 2];
  Real y[N,1];
equation
  for i in 1:N loop
    y[i,1] = u[i,1]*u[i,2];
  end for;
  annotation(PD2MO = {Vector, Vector});
end qss_multiplier_vec;
```

Listado 31: data/vector/qss_multiplier_vec.mo

```
model vector_pow2
  constant Real p[1] = {1};
  constant Integer N = integer(p[1]);
  Real u[N, 1];
  Real y[N, 1];
equation
  for i in 1:N loop
    y[i, 1] = u[i,1]*u[i,1];
  end for;
  annotation (PD2MO={Vector,Vector});
end vector_pow2;
```

Listado 32: data/vector/vector_pow2.mo

```
model hysteretic_vec
  constant Real p[5] = {1, 1, 1, 1, 1};
  parameter Real xl = p[1];
  parameter Real xu = p[2];
  parameter Real yl = p[3];
  parameter Real yu = p[4];
  constant Integer N = integer(p[5]);
  Real u[N, 1];
  Real y[N, 1];
  discrete Real state[N];
equation
  for i in 1:N loop
    y[i, 1] = pre(state[i]);
  end for;
algorithm
  when time>0 then
    for i in 1:N loop
      if u[i, 1] > xu then
        state[i] := yu;
      end if;
    end for;
    for i in 1:N loop
      if u[i, 1] < xl then
        state[i] := yl;
      end if;
    end for;
  end when;
  for i in 1:N loop
    when u[i, 1] > xu then
      state[i] := yu;
    end when;
  end for;
  for i in 1:N loop
    when u[i, 1] < xl then
      state[i] := yl;
    end when;
  end for;
  annotation (PD2MO={Vector,Vector});
end hysteretic_vec;
```

Listado 33: data/vector/hyst_vec.mo

```
model vector_sat
  constant Real p[3] = {-1, 1, 1};
  constant Integer N = integer(p[3]);
  parameter Real xl(fixed = true) = p[1];
  constant Real xu(fixed = true) = p[2];
  Real u[N, 1];
  Real y[N, 1];
  discrete Real under[N];
  discrete Real above[N];
initial algorithm
  for i in 1:N loop
    when time > 0 then
      if u[i, 1] < xl then
        under[i] := 1;
      else
        under[i] := 0;
      end if;
      if u[i, 1] > xl then
        above[i] := 1;
      else
        above[i] := 55;
      end if;
    end when;
  end for;
equation
  for i in 1:N loop
    y[i, 1] = pre(under[i]) *
                xl + (1 - pre(under[i])) *
                (pre(above[i]) *
                xu + (1 - pre(above[i])) *
                u[i, 1]);
  end for;
algorithm
  for i in 1:N loop
    when u[i, 1] < xl then
      under[i] := 1;
    end when;
    when u[i, 1] >= xl then
      under[i] := 0;
    end when;
    when u[i, 1] > xu then
      above[i] := 1;
    end when;
    when u[i, 1] <= xu then
      above[i] := 0;
    end when;
  end for;
  annotation(PD2MO = {Vector, Vector});
end vector_sat;
```


Listado 34: data/vector/vector_sat.mo