Sistemas empotrados y móviles

Desarrollo de Software Basado en Componentes y Servicios

M.I. Capel

ETS Ingenierías Informática y
Telecomunicación
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada
Email: manuelcapel@ugr.es

DSBCS

Máster en Ingeniería Informática

13 de diciembre de 2016

Índice

Índice

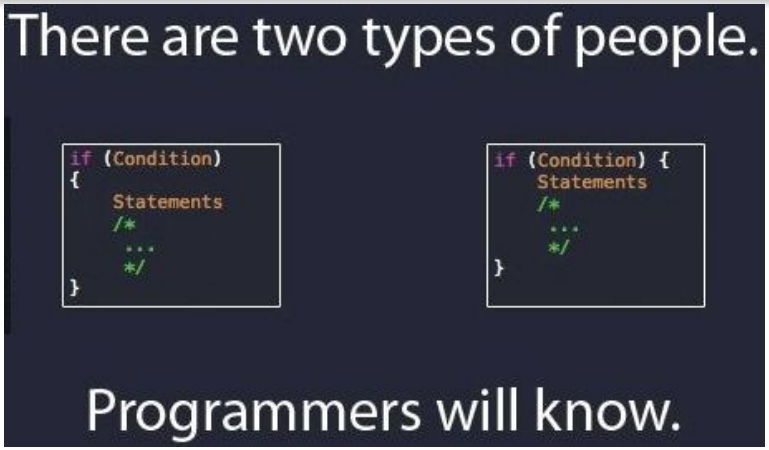
Introducción

- El desarrollo de software para empotrados necesita de un enfoque especial diferente del supuesto habitualmente cuando desarrollamos con Java/C++/Python. . .
- "No sirven más" las abstracciones habituales:*objetos*, ejecución concurrente basada en entrelazamientos, independencia de ejecución entre instrucciones atómicas, desprecio del tiempo de ejecución de cada sentencia, etc.
- El software de sistemas empotrados no se ejecuta en la "máquina idealizada" de Turing.
- El *empotrado* no está pensado (necesariamente) para ser una parte de un computador, tampoco su software.
- Cometido principal: *interaccionar con el "mundo real"*, lo que implica restricciones especiales.

Introducción – II

- Métodos de diseño para empotrados (“artesanales”) vs. los del software tradicional (“ingenieriles”).
- Necesidad de un nuevo *“conjunto de abstracciones”*, en orden de nivel de abstracción:
  - Modelos de computación, Metodologías
  - Lenguajes de programación, Constructos sintácticos
- Desideratum: (1) independencia de la plataforma, (2) reusabilidad (3) eficiencia, (4) confiabilidad, (5) predecibilidad, (6) robustez y tolerancia a fallas.
- Configuración actual en red de los “empotrados”(los DSP programables ahora se ejecutan con la pila TCP/IP)
- Adaptabilidad descargable desde el usuario, migración de código entre empotrados conectados a la red, etc.

Características de este software



- El software para empotrados ha de garantizar una serie de requerimientos *no funcionales*:
  - Puntualidad y medida del tiempo
  - Concurrencia
  - Vivacidad
  - Reactividad y soporte para heterogeneidad
- ¿Lenguajes síncronos (Esterel, Lustre, Signal, Argos . . .)?
- No se ha podido integrar satisfactoriamente el diseño orientado a objetos con los STR, hasta ahora

Componentización basada en *objetos* vs. *procesos*

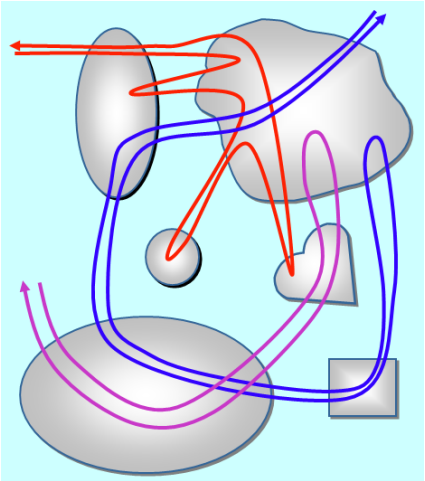


Figura: Los hilos (threads) entran y salen de los objetos sin tener en cuenta la estructura subyacente del programa

- Tecnologías basadas en componentes
- Tecnologías basadas en procesos
- El problema de la *composición*
- Propiedades temporales y descripción signatura de tipos
- Problemática de *análisis de concurrencia en ejecución*

Modelado basado en *actores*

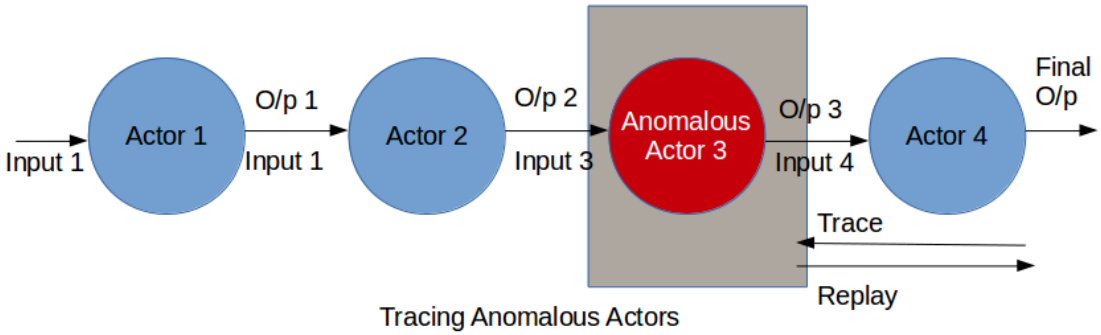


Figura: Modelo de actores mostrando un comportamiento anómalo

Definición de *actor* (Carl Hewitt, 1973)

- Unidad de computación, que encapsula:
  - Procesamiento
  - Estado
  - Comunicaciones
- Un actor es una entidad inherentemente concurrente

Modelado basado en *actores*–II

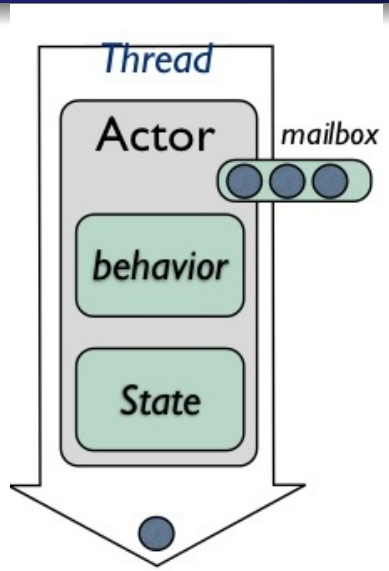


Figura: Estructura de un *actor*

- Modelo "Actor": mantener un estado interno mutable y comunicarse con otros actores mediante paso de mensajes asíncrono
- No se comparte el estado
- Los mensajes se mantienen en el *buzón de correo* y son procesados en orden

Modelado basado en *actores*–III

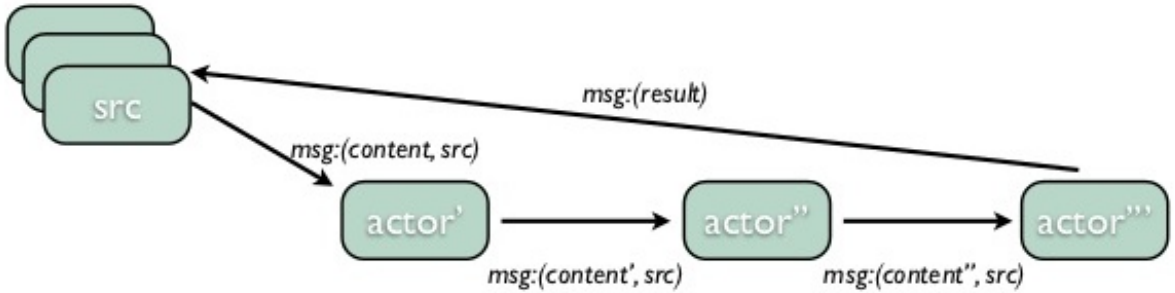


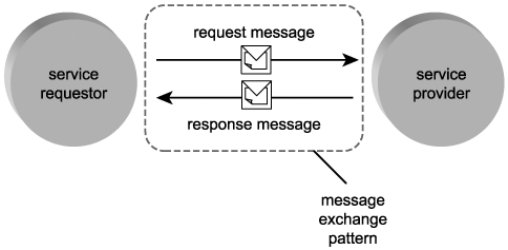
Figura: Comunicaciones entre actores

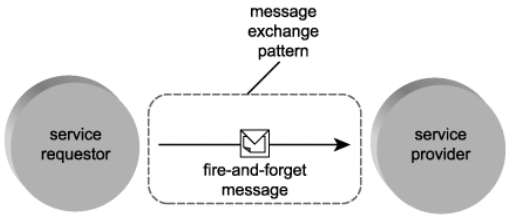
- Los actores en la práctica pueden:
  - Enviar un número finito de mensajes a otros actores
  - Manejar los mensajes recibidos
  - Crear un número finito de nuevos actores
  - Designar el comportamiento que van a seguir cuando reciban el próximo mensaje

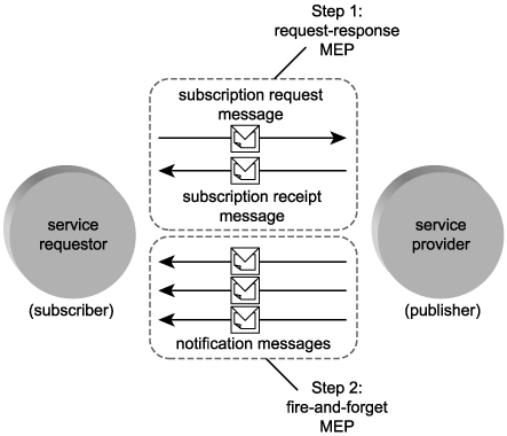
Programación con actores

```
1  //definir el protocolo de un actor
2  case object Incrementar
3  case object ConseguirCuenta
4
5  //definir el actor
6  class Contador extends Actor{
7    private var contador=0
8    def receive={
9      case Incrementar=>contador+=1 /
10     case:ConseguirCuenta=>self.reply(contador)
11   }
12 }
```

Tipos de comunicación







1. Disparar y esperar
2. Disparar y olvidar
3. Disparar y conseguir 1 *futuro*

Programación con actores II

```
1 import akka.actor.Actor
2 import akka.actor.ActorRef
3 //actorRef es la referencia al actor
4 val contador:ActorRef= Actor.actorOf(new Contador).start
5 //Enviar 1 mensaje a un actor; envio asincrono
6 Contador!Incrementar
7 //Enviar y esperar (hasta el timeout)
8 val valueOpt= (contador!!ConseguirCuenta).as[Int]
```

Enviar y devolver 1 futuro

```
1  val futuro= contador!!!ConseguirCuenta
2
3  futuro.await
4  val resultado= futuro.get
5
6  //Esperar porque hay mas
7  //Esperar el cierre cuando se haya completado
8  future.onComplete{
9    f=>println(f.get())
10 }
```

Modelado basado en *actores*–IV

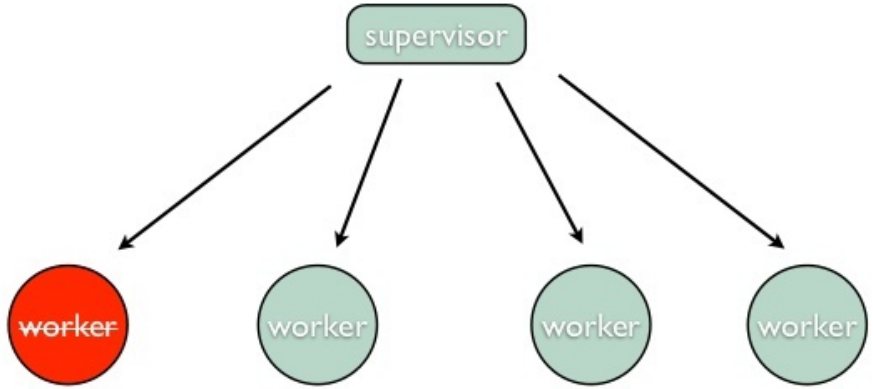


Figura: Tolerancia a fallos con el modelo *Actor*

Los actores en la práctica pueden:

- Funcionar en un contenedor
  - Registro de actores
  - Gestionar las asignaciones de actores a procesos
  - Tolerancia a fallos, supervision
- Estar distribuidos (implícitamente/Explícitamente)
- Se pueden intercambiar dinámicamente

Librerías y marcos de trabajo para el modelo Actor

Nombre	Última edición	Lenguajes
Akka Toolkit	2015	Java
CAF (C++ Actor Fram.)	2015	C++
Celluloid	2014	C++ , Ruby
Orbit	2015	Java
Orleans	2015	.NET
SObjectizer	2015	C++
Thespian	2015	Python

Arquitecturas de componentes basadas en una *"filosofía de actores"*

- Los *actores* interaccionan según un modelo concreto de comunicación
- Interfaces con declaraciones temporales, a diferencia de los modelos de componentes basados en *objetos*
- Estructuración de interfaces mediante un *sistema de tipos* con comprobación dinámica y en el diseño automática
- Enfoque de modelo *altamente composicional* y facilita construcciones concurrentes

Arquitecturas de componentes basadas en una *"filosofía de actores"*–II

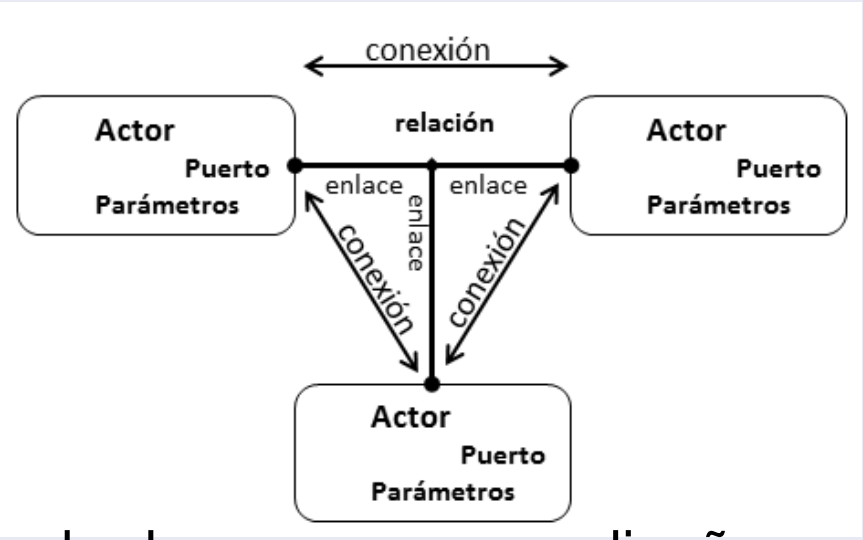
Sintaxis abstracta de un diseño

-Ensamblaje de empotrados como combinación componentes

-Sistemas de tipos vs. herencia/ subclasificación

-Parámetros y puertos

-Tiempo: construcción sintáctica nivel 1



-Sirve para definir cómo se puede descomponer un diseño en componentes interactuantes sin importarnos cómo se realizará la interconexión entre componenetes
-Podría definir una jerarquía de actores

Arquitecturas de componentes basadas en una *"filosofía de actores"*–III

Sintaxis concreta

- Ha de especificar el significado de la interconexión entre componentes
- La sintaxis abstracta y concretas definen una relación 1-a-muchos
- No se requiere forzosamente una sintaxis visual, pero siempre es una ayuda para comprender diseños de sistemas complejos
- Las sintaxis visuales pueden ser tan precisas y completas como las textuales,
  - VHDL, Verilog
  - FSM, StateDiagrams–UML
  - Vergil/Ptolemy ...

Sintaxis concreta desarrollada con un editor visual

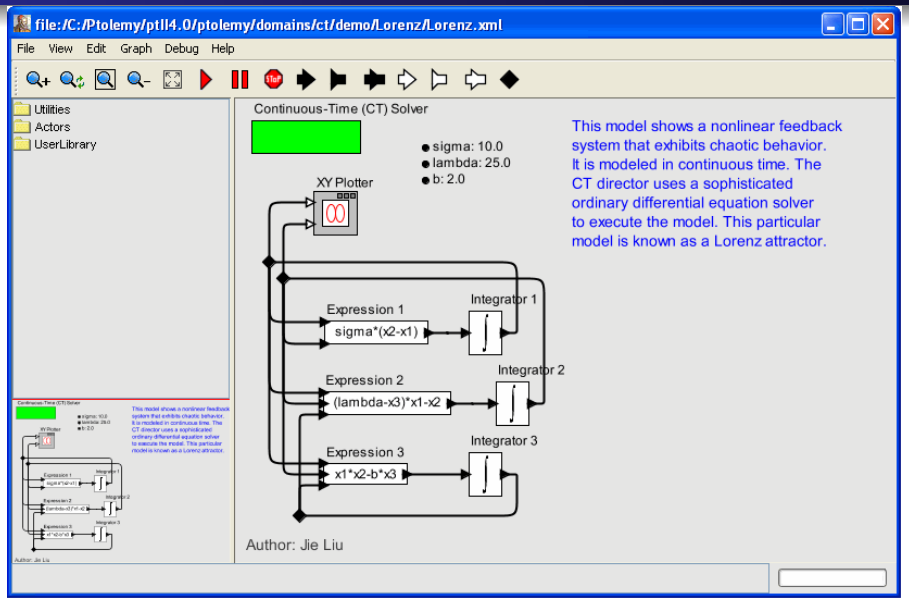


Figura: Ejemplo de sintaxis concreta visual, desarrollado con el editor visual Vergil que incluye el marco de trabajo Ptolemy II.

$$\frac{d\,x_1}{d\,t} = \sigma \times (x_2 - x_1)$$
$$\frac{d\,x_2}{d\,t} = (\lambda - x_3) \times x_1 - x_2$$
$$\frac{d\,x_3}{d\,t} = x_1 \times x_2 - \beta \times x_3$$

Sintaxis concreta desarrollada con un editor visual – II

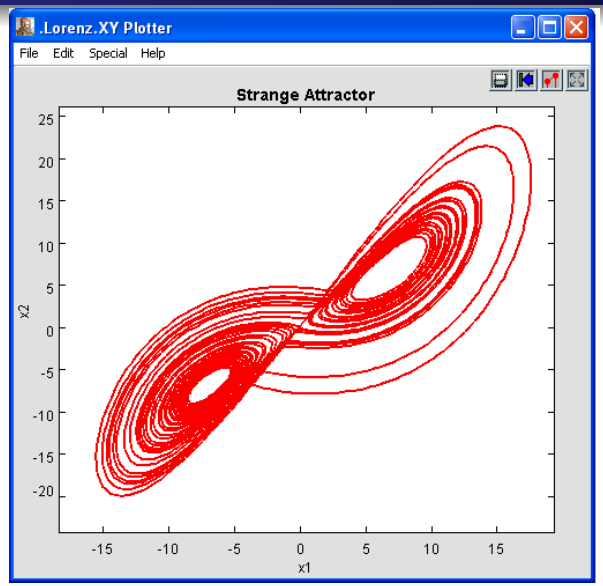


Figura: El modelo representa al “atractor de Lorenz”.

Atractor

- Un sistema no lineal de control con realimentación, cuya solución muestra un comportamiento caótico
- El reloj central (CT) de Ptolemy utiliza un modelo de bloques para resolver un sistema de 3 ecuaciones diferenciales ordinarias

Ideas fundamentales

Significado de la "*semántica*" en el contexto de modelos basados en "actores'

- Proporcionar significado a componentes y conexiones
- Ejemplos posibles:
  - Componente: proceso | estado
  - Conexión: comunicación entre componentes | transición
- Los modelos semáticos pueden ser también entendidos como *patrones arquitectónicos*
- Pero los denominaremos: "*Modelos de Computación*"

Modelos de computación últiles para el diseño de empotrados

- Gobiernan las interacciones entre componentes de 1 diseño
- Marco de trabajo conceptual para combinar componentes que representan empotrados
- Soportar la "*Concurrencia* de forma natural
  - No existe todavía un modelo de computación universal para representar la computación concurrente

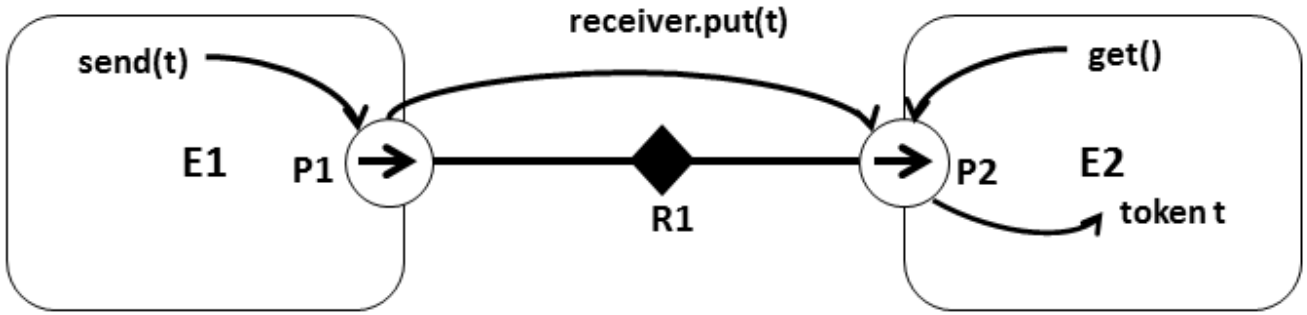
Modelos concurrentes

- Un diseño efectivo de un sistema concurrente necesita varios niveles de abstracción encima del "*hardware*"
- Las aplicaciones para empotrados se construyen para un determinado *modelo de computación*, tanto si el desarrollador es consciente como si no.
- Pluralidad de estilos para modelar la "Concurrencia":
  - Extensiones de lenguajes, en principio, no concurrentes
  - Occam (síncrono con comunicaciones no–determinísticas),
  - Esterel, Lustre y Argos soportan un modelo síncrono/reactivo
- Bajo nivel de aceptación por la Comunidad (de desarrolladores para empotrados): plataformas de ejecución limitadas

Modelos concurrentes II

- Utilización de lenguajes de coordinación
- Independencia entre el lenguaje de programación elegido y la selección del modelo de computación:
  - El lenguaje de programación SystemC sigue este enfoque
- Ejemplo de mezcla de lenguajes para un mismo diseño de sistema empotrado:
  - VHDL facilita el acceso a la FPGA para implementaciones en hardware
  - Java para conseguir transportabilidad
  - C proporciona une ejecución eficiente del código

Mecanismo de comunicación abstraido



- Los actores “productor” y “consumidor” envían/recuperan datos llamando a métodos definidos en 2 puertos
- Existe polimorfismo de métodos porque cómo reaccionarán a la llamadas dependerá del modelo de computación final
- Los modelos de computación pueden ser tan específicos como nos permita la plataforma: los *pipes* de UNIX no están predispuestos a evitar interbloqueos

Modelos de Computación

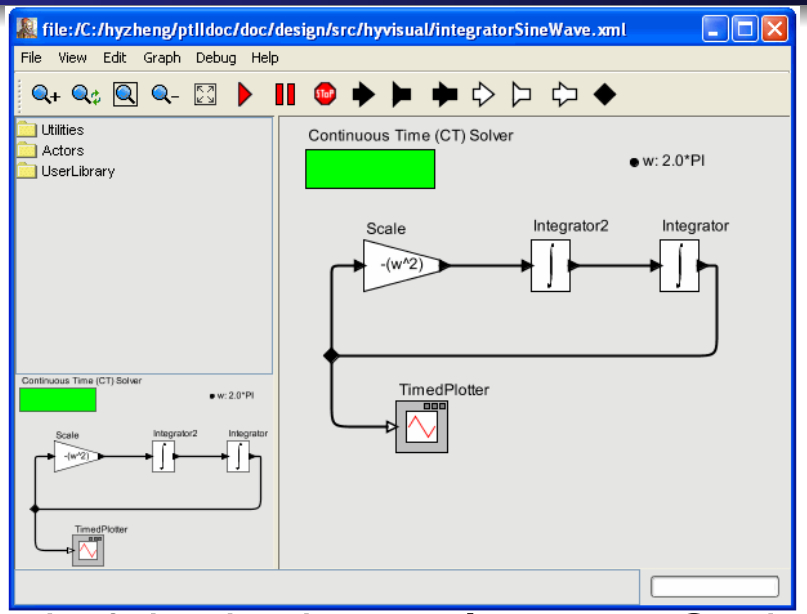


Figura: Un modelo de fujo de datos síncrono: Onda electromagnética que se propaga en el vacío

- Modelos basados en Flujo de Datos (DF): en los actores se programan instrucciones atómicas que se "*disparan*" cuando hay datos disponibles
- LabVIEW es un ejemplo de marco de trabajo comercial que usa este modelo.

Modelos de Computación – II

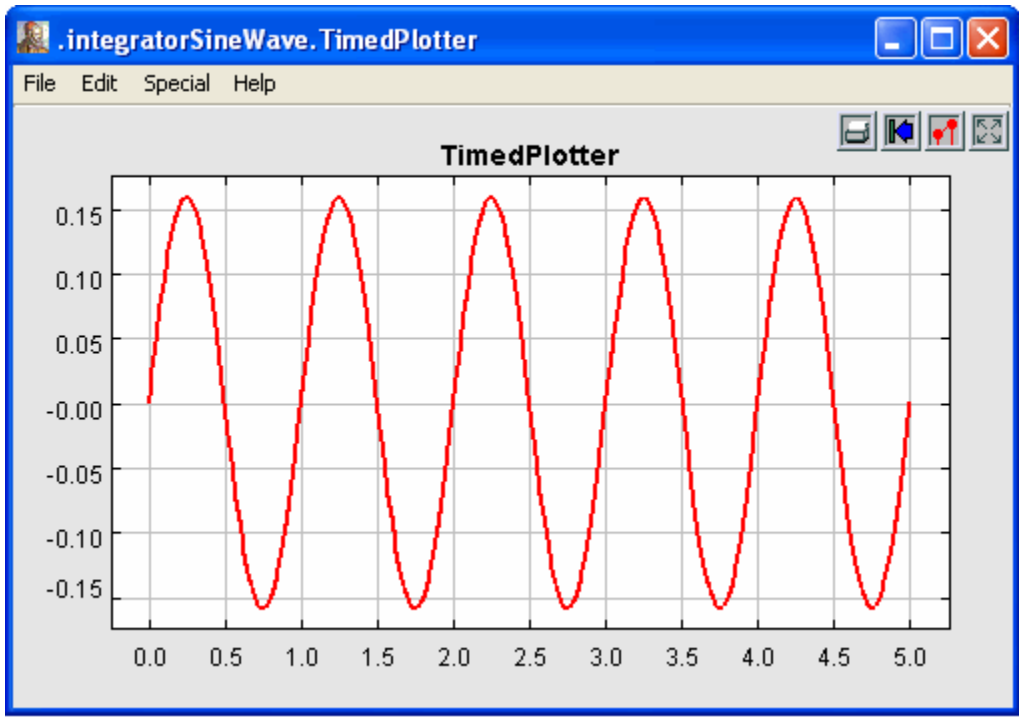


Figura: La solución al modelo de flujo de datos es una onda sinusoidal.

Modelos DF y SDF

- Sistemas empotrados que programan eventos temporizados dirigidos por relojes–hardware
- Periodos de tiempo fjjos, que se prestan a realizar *Análisis Monótono basado en Frecuencias de Ejecución* de tareas
- Posibilidad de obtener planificación estática de las tareas
- Abstracción–software dirigida por el tiempo, que resulta ser independiente del hardware.
- Combinado con máquinas de estados finitos (FSM) puede proporcionarnos modelos híbridos bastante descrtiptivos.
- En el sistema *Scenic*, los componentes son procesos que se ejecutan indefinidamente, se detienen para esperar los ticks de reloj o condiciones sincronizadas con el reloj.

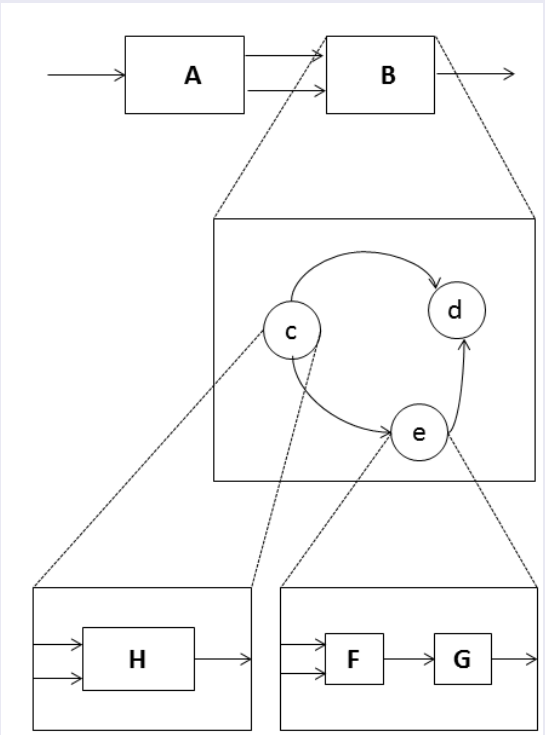
Composición jerárquica de un FSM con modelos de computación concurrentes

-Utilidad de combinar modelos de computación concurrentes con máquinas de estados finitos (FSM)

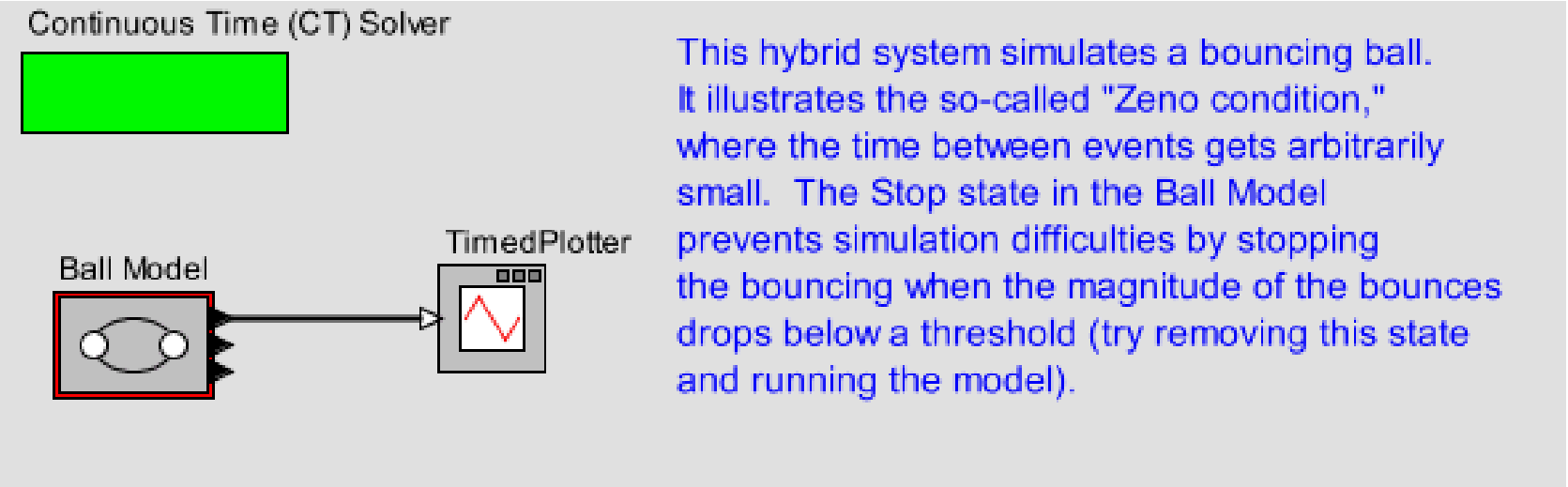
-Los componentes de 1 FSM se denominan estados (sólo 1 activo en cada instante)

-Las conexiones entre estados en un FSM se denominan transiciones

-Modelos *modales*

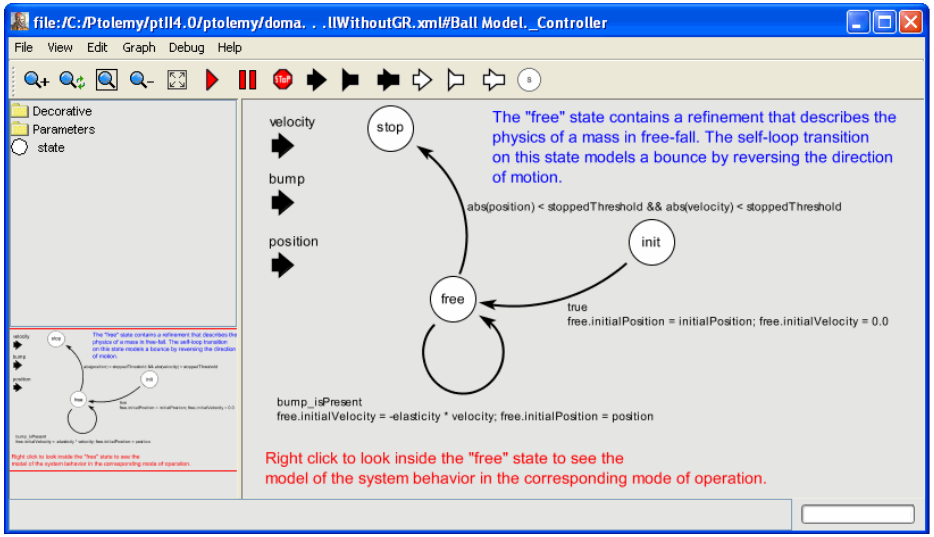


Sistema híbrido que modela una pelota que rebota en HyVisual



Para que el simulador, que utiliza un modelo contínuo de tiempo, no se bloquee se utiliza un FSM con un estado para parar la simulación cuando se da una condición de “Zenon”.

FSM que modela los estados de la pelota rebotante



- El estado “*free*” contiene el modelo físico que describe la aceleración de una masa que se encuentra en caída libre.
- El estado “*stop*” detiene la simulación cuando se da la condición de Zenon entre los puntos de rebote.

FSM que modela los estados de la pelota rebotante–II

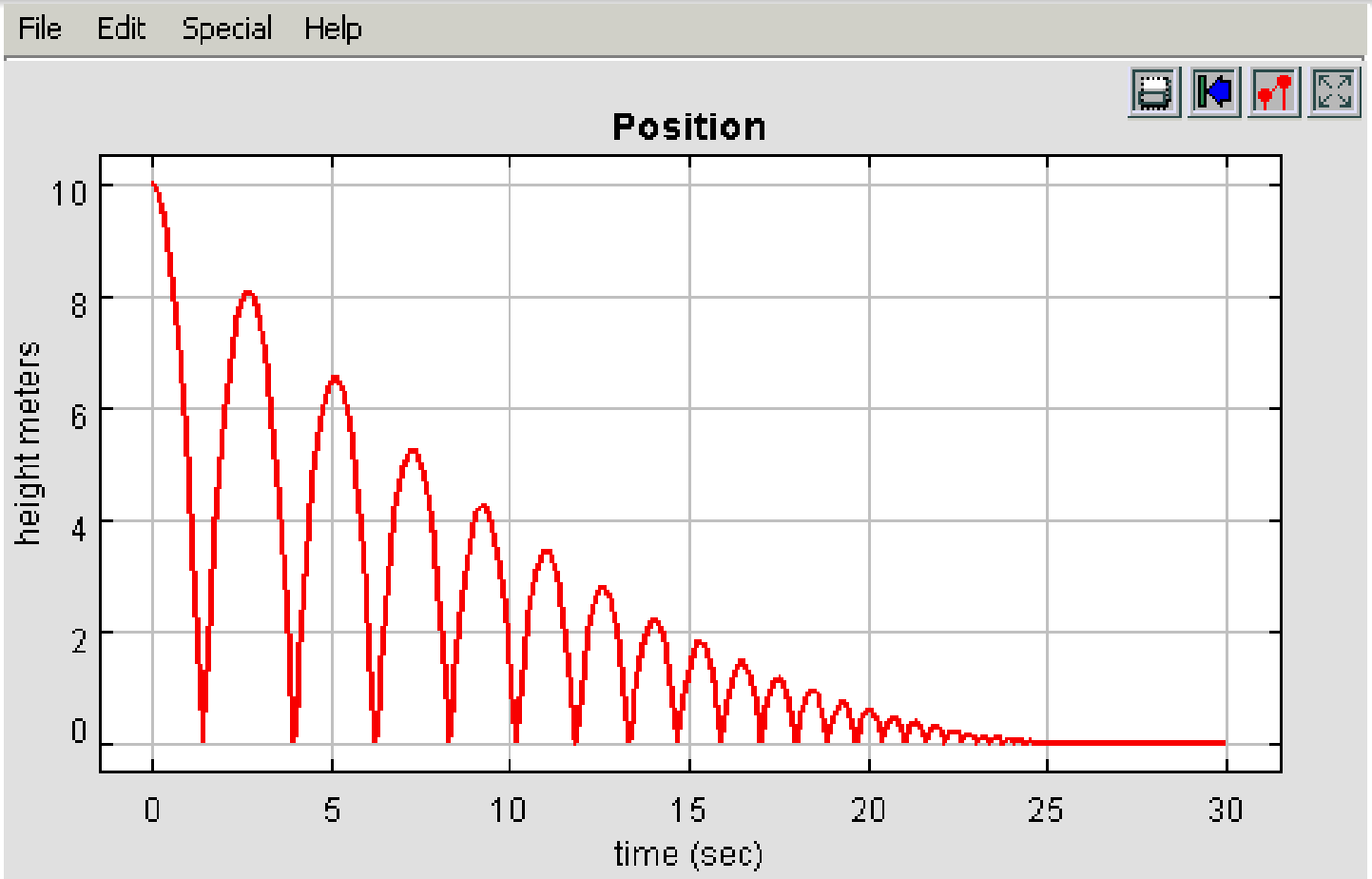


Figura: Solución del sistema de ecuaciones que muestra la disminución de la amplitud de los rebotes con el tiempo.