Simulación de Máquinas Finitas (Stateflow)

Materia: Simulación de procesos

Prof. D.Sc. BARSEKH-ONJI Aboud

Facultad de Ingeniería
Universidad Anáhuac México

27 de enero de 2026

Agenda

Sistemas Reactivos

¿Qué es un Sistema Reactivo?

Es un sistema cuyo comportamiento está dominado por su **interacción con el entorno**. No ejecuta un algoritmo de principio a fin, sino que reacciona a una secuencia de eventos que pueden ocurrir en cualquier momento y en cualquier orden.

Ejemplos

- El controlador de un robot industrial (agarrar, soltar, esperar).
- La lógica de una transacción en línea.
- Un sistema de control de vuelo.

Finite State Machines (FSM)

Definición

Una FSM es un modelo matemático de computación que se puede encontrar en exactamente uno de un número finito de **estados** en un momento dado. Puede cambiar de un estado a otro en respuesta a **eventos**; este cambio se denomina **transición**.

Una FSM se define por:

- Un conjunto de **estados** posibles.
- Un **estado inicial**.
- Un conjunto de **eventos** (o entradas).
- Un conjunto de **acciones** (o salidas).
- Una **función de transición** que define el siguiente estado.

Limitaciones de las FSM Clásicas

El Problema de la 'Explosión de Estados'

A medida que un sistema crece, el número de estados necesarios para describir todas las posibles condiciones puede volverse inmanejable.

- Los diagramas se vuelven planos, extensos y difíciles de entender.
- Las FSM clásicas carecen de mecanismos eficientes para representar:
  - La **jerarquía** (estados dentro de otros estados).
  - La **concurrencia** (procesos que ocurren al mismo tiempo).

Agenda

Stateflow: La Evolución de las FSM

De los Statecharts a Stateflow

Para superar las limitaciones de las FSM, David Harel introdujo los **statecharts**, un lenguaje visual que extiende las máquinas de estados. **Stateflow** es la implementación de MathWorks de este poderoso formalismo, integrado completamente en Simulink.

Características Clave de Stateflow

Stateflow añade a las FSM conceptos cruciales como:

- Jerarquía (Hierarchy)
- Paralelismo (Parallelism / Orthogonality)
- Historial (History Junctions)
- Comunicación con Eventos y Datos

Características Clave: Jerarquía y Paralelismo

Jerarquía (Hierarchy)

- Los estados pueden contener otros estados (subestados), creando **superestados**.
- Permite organizar la lógica de manera modular y por niveles de abstracción.
- **Ejemplo:** Un estado `Procesando_Orden` puede contener subestados como `Verificando_Pago` y `Asignando_Inventario`.

Paralelismo (Parallelism)

- Un superestado puede tener múltiples subestados activos **simultáneamente** (estados AND).
- Fundamental para modelar sistemas con modos de operación independientes y concurrentes.
- **Ejemplo:** En un vehículo, el sistema de transmisión y el de climatización operan en paralelo.

Características Clave: Historial y Datos

Historial (History Junctions)

- Permite que un superestado 'recuerde' el último subestado que estuvo activo antes de una transición de salida.
- Al reingresar al superestado, se puede reanudar la ejecución desde donde se quedó.
- Muy útil para modelar interrupciones.

Eventos y Datos

- Los diagramas de Stateflow se comunican con el modelo de Simulink a través de:
  - **Entradas:** Eventos y datos que activan transiciones y modifican la lógica.
  - **Salidas:** Señales y llamadas a funciones que afectan al resto del sistema.

Agenda

La Crisis de las Máquinas de Estados Convencionales

El Problema: La 'Explosión de Estados'

David Harel se propuso superar las limitaciones de las FSM tradicionales, que fallaban al describir sistemas complejos del mundo real. El principal problema era la falta de estructura, que llevaba a diagramas planos, masivos y visualmente caóticos.

La Solución de Harel

statecharts = state-diagrams + depth + orthogonality + broadcast-communication

Profundidad: Jerarquía y Agrupamiento

La primera innovación clave es la **profundidad**, que introduce el concepto de jerarquía en los estados.

Agrupamiento (Clustering)

Si varios estados tienen una transición común bajo el mismo evento, podemos agruparlos en un **superestado**. La transición se dibuja una sola vez desde el borde del superestado, economizando drásticamente el diagrama.

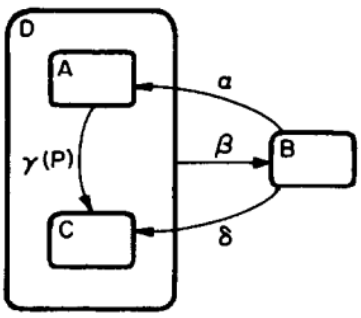


Figura 1: Agrupamiento de estados con una transición común.

Profundidad: Entradas por Defecto e Historial

Entrada por Defecto

Especifica a cuál de los subestados se debe entrar si la transición de entrada no lo indica explícitamente.

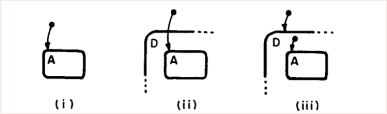


Figura 2: Transición por defecto.

Conector de Historial (H)

Permite que un superestado 'recuerde' el último subestado activo. Al reingresar, la ejecución se reanuda donde se quedó, lo cual es ideal para modelar interrupciones.

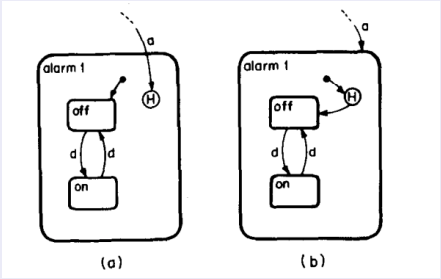


Figura 3: Conector de Historial.

Ejemplo Práctico: Alarma de Reloj Digital

En este ejemplo de Harel, los estados de sonido de la alarma se agrupan en el superestado `alarms-beep`.

Ventaja de la Jerarquía

En lugar de dibujar múltiples flechas de salida (una desde cada subestado), se dibuja una única transición desde el borde del superestado. La transición `any button pressed` se aplica sin importar en cuál de los subestados de alarma se encuentre el sistema, simplificando enormemente el diagrama.

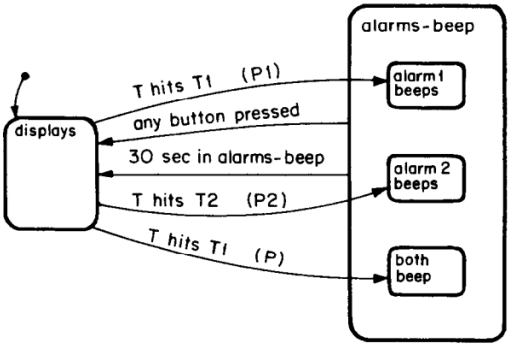


Figura 4: Agrupamiento de estados de alarma en un superestado.

Diagrama Completo: Reloj Citizen según Harel

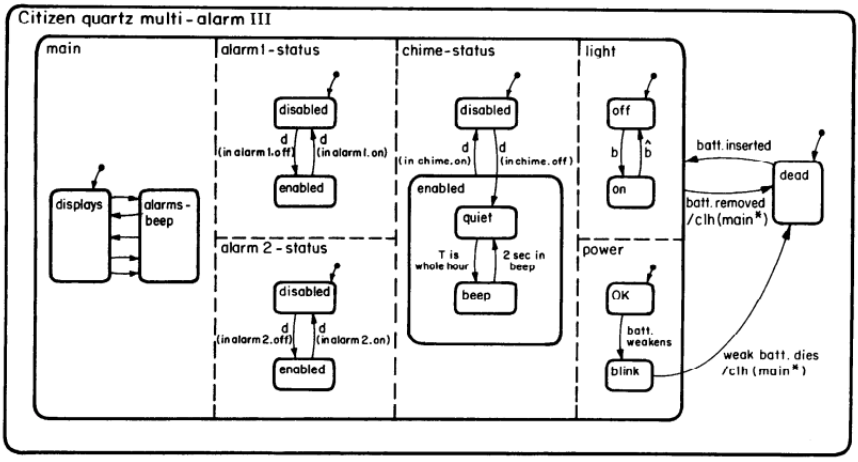


Figura 5: Diagrama de estado del sistema Citizen Quartz Multi-Alarm III.

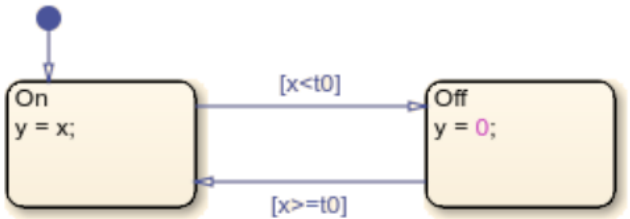
Agenda

Componentes Fundamentales: Estado y Transición

Estado (State)

Representado por un rectángulo con esquinas redondeadas, es un modo de operación del sistema. Puede tener acciones asociadas:

- `entry (en)`: Se ejecuta una vez al entrar al estado.
- `during (du)`: Se ejecuta en cada paso de tiempo mientras el estado está activo.
- `exit (ex)`: Se ejecuta una vez al salir del estado.



Componentes Fundamentales: Estado y Transición

Transición (Transition)

Representada por una flecha, indica un posible cambio de estado. Su etiqueta define la lógica:

`evento[condición]{acción}/acción`

- `evento`: El suceso que 'despierta' la transición.
- `[condición]`: Expresión booleana que debe ser verdadera.
- `{acción}`: Se ejecuta si la condición es verdadera.
- `/acción`: Se ejecuta al completar la transición.

Componentes Fundamentales: Flujo y Datos

Elementos de Flujo

- **Transición por Defecto:** Una flecha que indica cuál es el primer estado que se activa en un nivel jerárquico.
- **Conector de Unión (Junction):** Un círculo que sirve como punto de decisión para crear lógica condicional (if-then-else) en las transiciones.

Interfaz del Diagrama

- **Datos (Data):** Son las variables del diagrama (Input, Output, Local, Parámetros).
- **Eventos (Events):** Son disparadores explícitos e instantáneos que notifican que 'algo ha sucedido'.

Agenda

Modelando el Proceso de *Order Fulfillment*

La Columna Vertebral del Negocio

El proceso de cumplimiento de pedidos es la secuencia de pasos desde que se recibe un pedido hasta que el producto es entregado. Su eficiencia y precisión tienen un impacto directo en la satisfacción del cliente y los costos operativos.

Relevancia en B2C vs. B2B

Contexto B2C (e-commerce)

- La **experiencia del cliente** es primordial.
- Se esperan visibilidad total del pedido y entregas rápidas.
- El principal desafío es gestionar un **alto volumen** de pedidos pequeños y personalizados.
- Un fallo puede resultar en la pérdida de un cliente para siempre.

Contexto B2B (Negocio a Negocio)

- Las transacciones son de **mayor valor** y complejidad.
- La **fiabilidad y la precisión** son cruciales.
- Un retraso puede detener la línea de producción de un cliente, generando costos enormes.
- La integración con sistemas ERP es fundamental.

¿Por qué modelar este proceso con Stateflow?

Naturaleza Basada en Estados y Eventos

Una orden de compra no es un simple dato que fluye; es una entidad que posee un **estado** en todo momento ('Pendiente', 'Procesando', 'Enviado', 'Cancelado'). Este estado cambia en respuesta a **eventos** discretos ('pago_confirmado', 'inventario_insuficiente'). Esta naturaleza hace que Stateflow sea la herramienta ideal.

Con Stateflow, podemos:

- Representar visualmente todos los estados posibles de una orden.
- Definir explícitamente las condiciones y eventos que provocan un cambio de estado.
- Modelar lógicas de excepción complejas (pagos fallidos, falta de stock).
- Simular el proceso bajo diferentes escenarios para identificar cuellos de botella.
- Crear un modelo lógico que sirva como especificación para un sistema real.

Fase 1: El 'Camino Feliz'

Objetivo

Nos centraremos en el 'camino feliz' (*happy path*): un flujo de proceso lineal donde no ocurren excepciones. El objetivo es familiarizarnos con la creación de estados, transiciones y la interacción básica entre Simulink y Stateflow.

Estados del Proceso Básico

- `Inactivo` → `Recibida` → `Pago_Verificado` → `Inventario_Asignado` → `Orden_Enviada` → `Orden_Entregada`

Paso 1: Crear el Tipo de Dato Enumerado

¿Por qué usar una enumeración?

Para mejorar la legibilidad y robustez del modelo, usamos un tipo de dato enumerado para representar los estados. Esto nos permite usar nombres descriptivos (como `Recibida`) en lugar de números (1, 2, 3).

Paso 1: Crear el Tipo de Dato Enumerado

Código: `OrdenStatus.m`

Cree un archivo `.m` con el mismo nombre que la clase y guarde este código:

```
classdef OrdenStatus < Simulink.IntEnumType
  % Enumeracion para los estados de una orden.
  enumeration
      Inactivo(0)
      Recibida(1)
      Pago_Verificado(2)
      Inventario_Asignado(3)
      Orden_Enviada(4)
      Orden_Entregada(5)
  end
end

```

Paso 2 y 3: Configurar Simulink y las Variables de Stateflow

Modelo en Simulink

Se construye la estructura principal que se comunicará con el *Chart* de Stateflow.

- 5 bloques `Constant` para simular las entradas de control.
- 1 bloque `Stateflow Chart` para la lógica.
- 1 bloque `Display` para ver el estado actual.

Variables (Símbolos) en Stateflow

Se define la interfaz de datos del *chart*.

- 5 variables de **entrada** (booleanas) para las condiciones: `nueva_orden`, `pago_ok`, etc.
- 1 variable de **salida** (del tipo `OrdenStatus`) para reportar el estado: `estado_orden`.

Paso 2 y 3: Configurar Simulink y las Variables de Stateflow

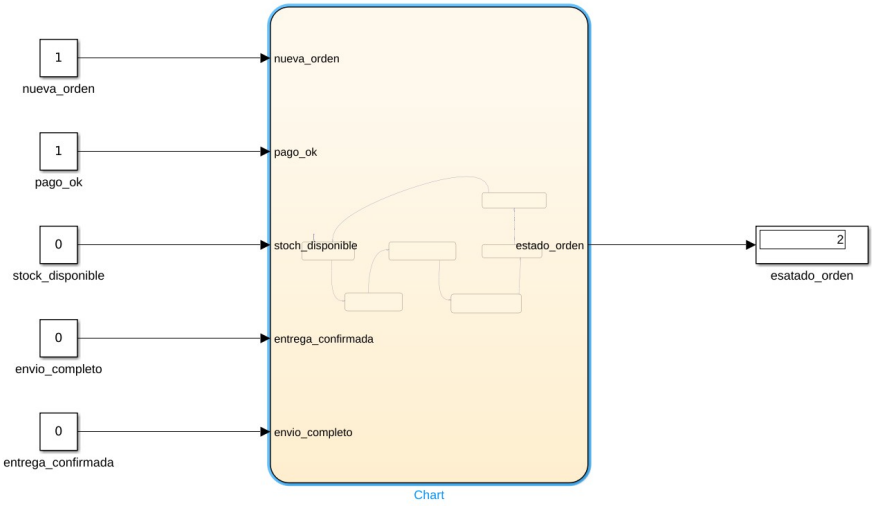


Figura 7: Panel de Símbolos con las variables.

Paso 4: Construir el Diagrama de Estados

Lógica del 'Camino Feliz'

Se dibujan los estados y se conectan con transiciones.

- Cada estado tiene una acción de entrada (`entry`) que actualiza la variable de salida `estado_orden`.
- Cada transición está 'protegida' por una condición que debe ser verdadera para que el sistema avance al siguiente estado (e.g., `[pago_ok == 1]`).

Paso 4: Construir el Diagrama de Estados

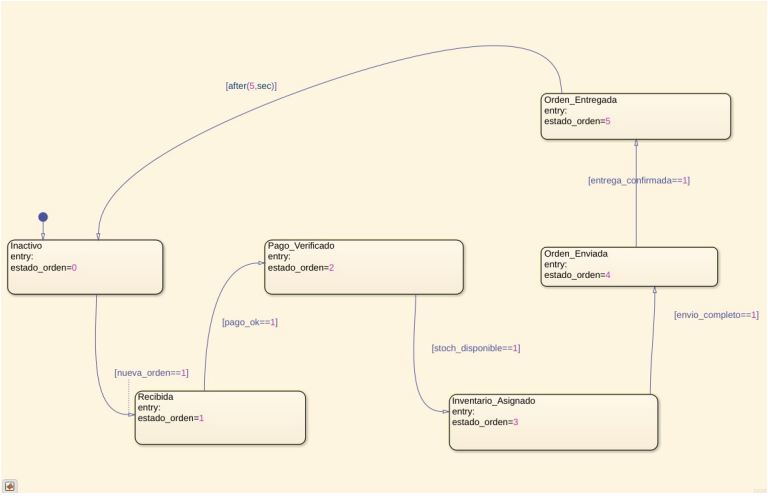


Figura 8: Diagrama de Stateflow completo para el modelo básico.

Paso 5: Simular y Verificar el Modelo

Prueba Interactiva

1. Configure el tiempo de simulación en `inf` e inicie la simulación. El `Display` mostrará `0` (Inactivo).
2. Durante la simulación, haga doble clic en el bloque `Constant` `nueva_orden` y cambie su valor a `1`.
3. Observe cómo el *chart* se anima, la transición se activa y el `Display` cambia a `1` (Recibida).
4. Repita el proceso para las demás entradas en secuencia para hacer que la orden avance por todos los estados hasta ser entregada.

Fase 2: Manejo de Excepciones y añadir realismo

Objetivo

El 'camino feliz' es poco realista. Haremos nuestro modelo más robusto al introducir dos excepciones comunes: un **fallo en el pago** y la **falta de inventario**.

Herramienta Clave: Conector de Unión (Junction)

Utilizaremos el conector de unión, un pequeño círculo que nos permite implementar lógica de decisión tipo **if-then-else** de manera gráfica para crear bifurcaciones en el flujo del proceso.

Paso 1: Actualizar el Tipo de Dato Enumerado

Nuevos Estados de Excepción

Debemos añadir los nuevos estados a nuestra clase `OrdenStatus.m` para poder referenciarlos en el modelo.

Paso 1: Actualizar el Tipo de Dato Enumerado

Código Actualizado: OrdenStatus.m

```
classdef OrdenStatus < Simulink.IntEnumType
  enumeration
      Inactivo(0)
      Recibida(1)
      Pago_Verificado(2)
      Inventario_Asignado(3)
      Orden_Enviada(4)
      Orden_Entregada(5)
      % --- Nuevos estados de excepcion ---
      Pago_Fallido(6)
      Fuera_de_Stock(7)
  end
end

```

Paso 2: Modificar el Diagrama de Estados con Uniones

Lógica de Decisión

- Se añaden los estados `Pago_Fallido` y `Fuera_de_Stock`.
- Se elimina la transición directa entre `Recibida` y `Pago_Verificado`.
- Se inserta un **conector de unión**.
- Desde la unión, una transición va a `Pago_Verificado` con la condición `[pago_ok == 1]`.
- Otra transición va a `Pago_Fallido` con la condición `[pago_ok == 0]`.
- Se repite un proceso similar para la decisión del inventario.

Paso 2: Modificar el Diagrama de Estados con Uniones

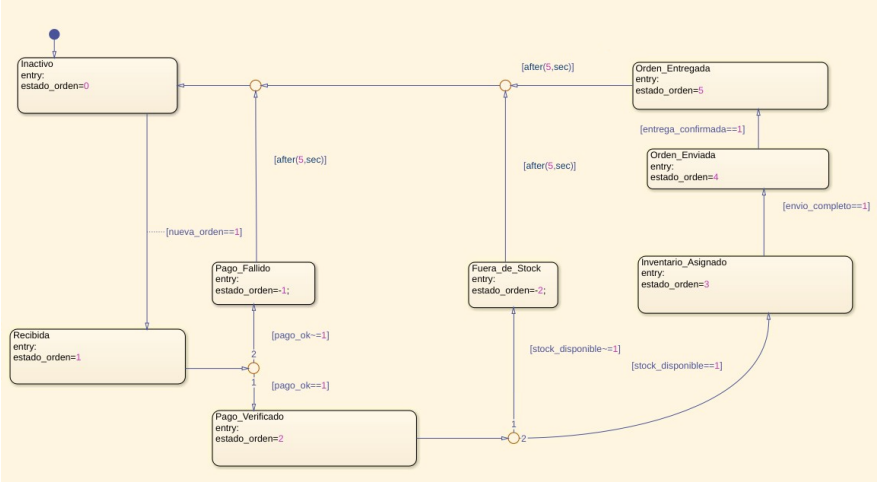


Figura 9: Diagrama con lógica de decisión para manejar excepciones.

Paso 3: Simular los Escenarios de Excepción

Prueba de Fallo de Pago

1. Activar `nueva_orden = 1`. El sistema transita a `Recibida`.
2. Asegurarse de que `pago_ok = 0`.
3. En el siguiente paso, el sistema tomará el camino hacia `Pago_Fallido`.
4. Después de 5 segundos, la orden se cancela y el sistema vuelve a `Inactivo`.

Prueba de Falta de Stock

1. Activar `nueva_orden = 1` y `pago_ok = 1`. El sistema llega a `Pago_Verificado`.
2. Asegurarse de que `stock_disponible = 0`.
3. El sistema tomará la ruta hacia `Fuera_de_Stock` y luego volverá a `Inactivo`.

Fase 3: Añadiendo Complejidad con Jerarquía y Paralelismo

Objetivo

Los procesos logísticos reales rara vez son secuenciales. Usaremos dos conceptos avanzados de Stateflow para modelar subprocesos que ocurren de forma concurrente:

- **Jerarquía (Superestados):** Para agrupar la lógica principal del procesamiento de la orden.
- **Paralelismo (Estados Ortogonales):** Para modelar la cancelación de una orden o el proceso de devolución, que pueden ocurrir en paralelo al flujo principal.

Paso 1: Actualizar Enumeración y Variables

Nuevos Estados

Añadimos estados para la cancelación y el ciclo de devolución a `OrdenStatus.m`.

```
% ... (estados anteriores)
Cancelada(8)
Devolucion_Solicitada(9)
Paquete_Recibido_Dev(10)
Reembolso_Procesado(11)

```

Nuevas Variables

Añadimos nuevas variables de **entrada** y **salida** al *chart* para gestionar los nuevos procesos.

- `solicitud_cancelar`
- `solicitud_devolver`
- `paquete_devuelto`
- `estado_devolucion`

Paso 2: Reestructurar con Jerarquía y Paralelismo

Crear un Superestado

- Se crea un estado grande `Orden_Activa` y se arrastran todos los estados del proceso principal (excepto `Inactivo`) dentro de él.
- Una única transición desde el **borde** de este superestado hacia un nuevo estado `Cancelada` permite interrumpir *cualquier* subestado activo con la condición `[solicitud_cancelar == 1]`.

Añadir Paralelismo

- El superestado `Orden_Activa` se descompone en dos regiones paralelas (AND) con una línea discontinua.
- Una región mantiene el flujo principal del pedido.
- La otra región contiene una nueva máquina de estados para gestionar el proceso de **devolución**, que se activa solo cuando el proceso principal llega al estado `Orden_Entregada`.

Diagrama Final del Modelo

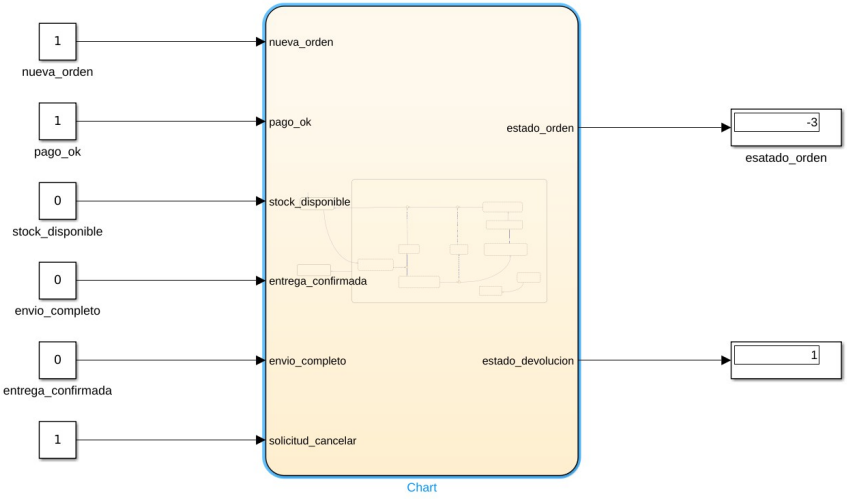


Figura 10: Diagrama de Stateflow final con jerarquía y estados paralelos.

Diagrama Final del Modelo

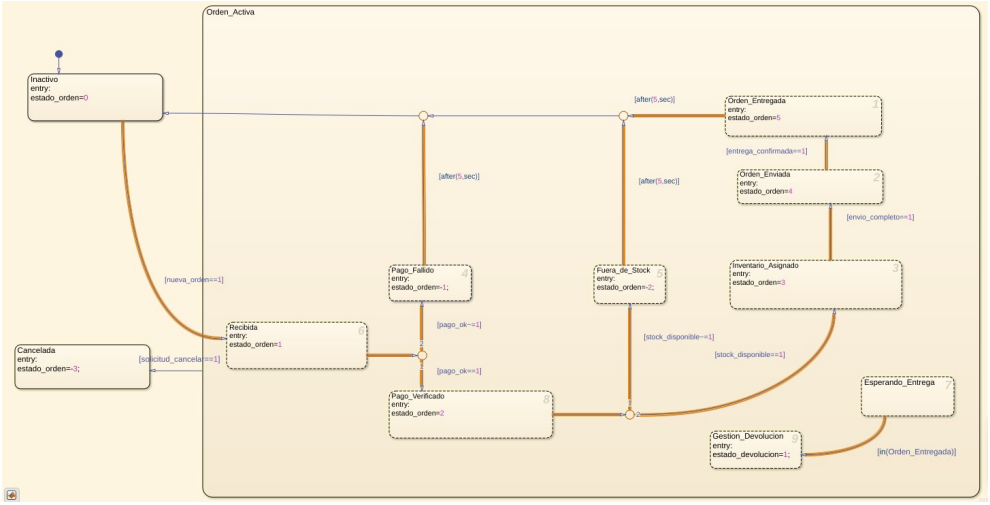


Figura 11: Diagrama de Stateflow final con jerarquía y estados paralelos.

Paso 3: Simular Escenarios Avanzados

Simular una Cancelación

1. Iniciar el proceso de una orden hasta un estado intermedio (e.g., `Inventario_Asignado`).
2. Activar la señal `solicitud_cancelar = 1`.
3. Observar cómo la ejecución salta inmediatamente del subestado activo al estado `Cancelada`, demostrando el poder de la transición de alto nivel.

Simular una Devolución

1. Completar el flujo principal hasta que la orden llegue a `Orden_Entregada`.
2. En este punto, la máquina de estados paralela de devolución se activa.
3. Activar `solicitud_devolver = 1`.
4. Observar cómo el subproceso de devolución avanza por sus propios estados, mientras el estado principal de la orden permanece en `Orden_Entregada`.