SASE 2011
Simposio Argentino de Sistemas Embebidos

Diagrama de Estado: Aplicaciones en sistemas embebidos

Ing. Juan Manuel Cruz (jmcruz@hasar.com)

Profesor Invitado

Seminario de Electrónica: Sistemas Embebidos (Cód: 66.48)

Ingeniería en Electrónica – FI – UBA

Buenos Aires, 4 de Marzo de 2011

Temario

- Introducción
- Estado del arte
- Problemática general
- Diagrama de Estado
- Paso a Paso
- Un ejemplo de aplicación

Introducción

- Cada vez es más frecuente el uso de modelos para describir el software de los sistemas embebidos. Los modelos no están pensados para visualizar código sino para representar un sistema con un nivel de abstracción superior al de los lenguajes de programación
- En cierta forma, la migración de una metodología de programación basada el lenguaje C hacia el desarrollo de software basado en modelos, supone un incremento en nivel de abstracción y en productividad similar al producido al cambiar desde lenguaje ensamblador hacia lenguaje C
- Los modelos ayudan a comprender el sistema a diseñar y favorecen el intercambio de ideas entre los miembros del equipo de diseño y sus clientes, además, gracias a la existencia de programas informáticos específicos

Introducción

- Los modelos permiten **simular** el funcionamiento de un sistema desde las primeras etapas del diseño y generar automáticamente el **código** fuente y la **documentación** del proyecto, manteniendo en todo momento el sincronismo entre modelo, código y documentación
- Sin embargo, en nuestro entorno, la práctica del uso de modelos para el desarrollo de software no está muy extendida, por ello el objetivo de este trabajo es proporcionar un **ejemplo práctico** a nuestros alumnos y colegas, **que impulse el despegue de esta metodología de programación**
- Fuente: Desarrollo de Software Basado en Modelos para Sistemas Embebidos - Mariano Barrón Ruiz

Estado del arte

- Un **modelo** es una **representación simplificada** de un sistema que contempla las propiedades importantes del mismo desde un determinado punto de vista. El uso de modelos es una actividad arraigada en técnicos e ingenieros y probablemente tan antigua como la propia ingeniería. Los modelos de mayor utilidad se caracterizan por ser:
  - **Abstractos.** Enfatizan los aspectos importantes del sistema y eliminan los aspectos irrelevantes
  - **Comprensibles.** Expresados en forma fácilmente perceptible por los observadores
  - **Precisos.** Representan fielmente el sistema modelado
  - **Predictivos.** Pueden utilizarse para responder cuestiones sobre el sistema modelado
  - **Económicos.** Mucho más baratos de construir y estudiar que el propio sistema

Estado del arte

- Los modelos empleados para crear software para sistemas embebidos, además de servir para lograr un conocimiento más profundo del problema, también se utilizan como elementos de entrada para la generación de código. La mayoría de los enfoques actuales en el desarrollo de software basado en modelos coinciden en:
  - Utilizar una **representación gráfica** del sistema a desarrollar
  - Describir el sistema con un cierto grado de **abstracción**
  - Generar **código** ejecutable para el sistema embebido **partiendo del propio modelo**
- Las máquinas de estados finitos constituyen una herramienta gráfica que ha sido utilizada tradicionalmente para modelar el comportamiento de sistemas electrónicos e informáticos. Una máquina de estados es un modelo computacional, basado en la teoría de autómatas, que se utiliza para describir sistemas cuyo comportamiento depende de los eventos actuales y de los eventos que ocurrieron en el pasado

Estado del arte

- En cada instante de tiempo la máquina se encuentra en un estado, y dependiendo de las entradas, actuales y pasadas, que provienen del ambiente, la máquina cambia o no cambia de estado, pudiendo realizar acciones que a su vez influyen en el ambiente
- Las máquinas de estados tradicionales son una excelente herramienta para el tratamiento de problemas sencillos o de mediana dificultad, pero su utilidad disminuye cuando se abordan sistemas más complejos
- A mediados de la década de 1980, David Harel - propuso una amplia extensión al formalismo convencional de las máquinas y diagramas de estados a la que denominó **statecharts**
- El término, según palabras de su autor, fue elegido por ser una combinación no utilizada del las palabras "flow" o "state" con "diagram" o "chart"

Estado del arte

- El objetivo principal del nuevo formalismo gráfico era **modelizar** o describir sistemas reactivos complejos
- El término reactivo se aplica a objetos que responden dinámicamente a los eventos de interés que reciben, y cuyo comportamiento lo define el orden de llegada de esos eventos
- Constituyen ejemplos de sistemas reactivos: los cajeros automáticos, los sistemas de reservas de vuelos, los sistemas embebidos en aviones y automóviles, los sistemas de telecomunicaciones, los sistemas de control, etc.
- Los statecharts extienden los diagramas de transición de estados convencionales con tres elementos principales: jerarquía, concurrencia y comunicación

Estado del arte

- El uso de jerarquías favorece el tratamiento de los sistemas con diferentes niveles de detalle; la concurrencia, también llamada ortogonalidad y paralelismo, posibilita la existencia de tareas independientes o con escasa relación entre ellas, y la comunicación permite que varias tareas reaccionen ante un mismo evento o envíen mensajes hacia otras tareas.
- El uso de los statecharts de Harel se ha extendido considerablemente tras convertirse en uno de los diagramas integrados en el UML (Unified Modeling Language) para describir el comportamiento de sistemas o de modelos abstractos
- El UML considera los diagramas gráficos como vistas o representaciones parciales del modelo de un objeto; los diagramas de UML representan tres vistas distintas del modelo: la vista de sus necesidades funcionales, la vista de su estructura y la vista de su comportamiento

Estado del arte

- La versión 2.0 de UML contempla el uso de hasta 13 tipos de diagramas que enfatizan diversos aspectos de la estructura, el comportamiento y la interacción entre las partes de un sistema
- Este elevado número de diagramas sirve como argumento a las críticas que algunos autores hacen al UML, los cuales consideran los statecharts de Harel como método valioso para representar procesos secuenciales, pero reprochan a los responsables del UML el no haber realizado una labor de síntesis de sus muchos años de experiencia, y haber fusionado conceptos y notaciones no del todo compatibles, produciendo un monstruo sobrecargado de elementos que se solapan y confunden a los usuarios

Problemática general

- Aunque es posible modelizar sistemas reactivos sin la ayuda de herramientas CASE, tal como propone el autor Miro Samek, lo cierto es que estas herramientas facilitan el trabajo al aportar sus cómodos editores gráficos; la posibilidad de disponer rápidamente de un modelo claro y ejecutable que permita la simulación temprana del comportamiento del sistema; la verificación funcional del modelo; la generación automática de código C, C++, o código en otros lenguajes de alto nivel; la generación automática de documentación; el seguimiento del grado de cumplimiento de las especificaciones; etc. Son numerosas las herramientas libres o comerciales disponibles, existiendo incluso herramientas de código abierto como UML StateWizard

Problemática general

- Una herramienta comercial especialmente adaptada al diseño de sistemas embebidos es IAR visualSTATE; se trata de una herramienta relativamente sencilla de aprender (sólo incluye uno de los 13 diagramas UML, el de los statecharts), que viene complementada por otras utilidades (verificador, simulador, generador de código C, etc.) muy importantes en el diseño de sistemas embebidos, que genera un código tan compacto que permite alojar sistema de mediana complejidad en microcontroladores de 8 bits con solo 2KB de memoria, que incrementa notablemente la productividad de los programadores, y cuya última consecuencia es mejorar la calidad del software generado. Todo lo anterior y la disponibilidad de una versión demo han sido las razones por las que hemos elegido IAR visualSTATE como herramienta para el desarrollo de software basado en modelos para sistemas embebidos. La versión demo tiene la misma funcionalidad que la versión comercial, pero limitada a 20 estados, que resultan suficientes para su uso en Escuelas

Problemática general

- IAR visualSTATE es un entorno gráfico para diseño, verificación e implementación de sistemas embebidos basados en máquinas de estados jerárquicas o statecharts. Entre sus características destacan:
  - Un entorno de desarrollo integrado que incluye un editor gráfico, herramientas de verificación y simulación, generador automático de código C y/o C++, y generador automático de documentación
  - Diseño gráfico de máquinas de estados jerárquicas basado en el subconjunto UML-Statechart
  - Verificación formal o matemática del modelo para hallar propiedades no deseadas del diseño: estados sin salida, estados inalcanzables, etc.
  - Herramienta de simulación o validación que permite, desde las primeras etapas del diseño, verificar que la aplicación se comporta de la forma deseada
  - Generador automático de código C/C++. El código generado es muy compacto y conforme al 100% con el modelo validado
  - Generador automático de documentación en formato RTF o HTML

Problemática general

- Los statecharts ayudan a construir modelos gráficos que describen con precisión el comportamiento de los sistemas
- Los modelos creados no guardan relación con el lenguaje de programación utilizado en el desarrollo de la aplicación, sin embargo, si mantienen una relación muy estrecha con el funcionamiento deseado del sistema: ésta relación, unida a la representación gráfica del modelo, facilita la comunicación y el intercambio de ideas entre los clientes y el equipo de desarrollo del sistema, independientemente del tipo de formación que posean
- Un modelo permite simular y visualizar la aplicación desde las primeras etapas del diseño sin necesidad de construir un prototipo hardware, lo cual facilita la eliminación de errores desde el principio

Problemática general

- Los programadores deben de cambiar la forma tradicional en la que abordan la tarea de desarrollo de software; deben trasladar su forma de pensar al dominio de la aplicación y liberarse de las exigencias impuestas por el lenguaje de programación utilizado
- Si la herramienta de modelado dispone de generadores automáticos de código y de documentación los beneficios son aún mayores, ya que los cambios que se realizan en el modelo se trasladan automáticamente al código generado y a la documentación creada, por lo que la propia herramienta se encarga de mantener el sincronismo entre el modelo, el código y la documentación
- El hecho de disponer de documentación actualizada es un aspecto de enorme importancia ya que facilita el mantenimiento de las aplicaciones

Diagrama de Estado

- Los diagramas de estado muestran el conjunto de estados por los cuales pasa un objeto durante su vida en una aplicación en respuesta a **eventos** (por ejemplo, mensajes recibidos, tiempo rebasado o errores), junto con sus respuestas y acciones. También ilustran qué eventos pueden cambiar el estado de los objetos de la clase.
- Normalmente contienen: **estados** y **transiciones**. Como los estados y las transiciones incluyen, a su vez, **eventos**, **acciones** y **actividades**, vamos a ver primero sus definiciones.
- Al igual que otros diagramas, en los diagramas de estado pueden aparecer **notas** explicativas y restricciones.

Eventos

- Un **evento** es una ocurrencia que puede causar la transición de un estado a otro de un objeto. Esta ocurrencia puede ser una:
  - **Condición** que toma el valor de verdadero (normalmente descrita como una expresión booleana). Es un **EventoCambio**
  - Recepción de una **señal** explícita de un objeto a otro. Es un **EventoSeñal**
  - Recepción de una **llamada** a una operación. Es un **EventoLlamada**
  - Paso de cierto **período de tiempo**, después de entrar al estado actual, o de cierta hora y fecha concretas. Es un **EventoTiempo**
- El nombre de un evento tiene alcance dentro del paquete en el cual está definido y puede ser usado en los diagramas de estado por las clases que tienen visibilidad dentro del paquete. Un evento no es local a la clase donde está declarado

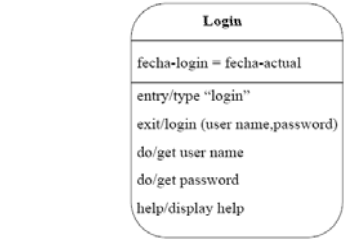
Acciones & Actividades

- Una **acción** es una operación atómica, que no se puede interrumpir por un evento y que se ejecuta hasta su finalización. Una acción puede ser:
  - Una llamada a una operación (al objeto al cual pertenece el diagrama de estado o también a otro objeto visible)
  - La creación o la destrucción de otro objeto
  - El envío de una señal a un objeto
- Cuando un objeto está en un estado, generalmente está esperando a que suceda algún evento. Sin embargo, a veces, queremos modelar una **actividad** que se está ejecutando. Es decir, mientras un objeto está en un estado, dicho objeto realiza un trabajo que continuará hasta que sea interrumpido por un evento
- Por lo tanto, una acción contrasta con una actividad, ya que ésta última puede ser interrumpida por otros eventos

Estados

- Un **estado** identifica una condición o una situación en la vida de un objeto durante la cual satisface alguna condición, ejecuta alguna actividad o espera que suceda algún evento. Un objeto permanece en un estado durante un tiempo finito (no instantáneo)
  - Un estado se representa gráficamente por medio de un rectángulo con los bordes redondeados y con tres divisiones internas. Los tres compartimentos alojan el nombre del estado, el valor característico de los atributos del objeto en ese estado y las acciones que se realizan en ese estado, respectivamente. En muchos diagramas se omiten los dos compartimentos inferiores
  - En el primer ejemplo viene representado el estado **Login** junto con sus tres divisiones. Asimismo, los diagramas de estado tienen un punto de comienzo, el **estado inicial**, que se dibuja mediante un círculo sólido relleno, y un (o varios) punto de finalización, el **estado final**, que se dibuja por medio de un círculo conteniendo otro más pequeño y relleno (es como un ojo de toro). Dichos estados, inicial y final, aparecen marcados en el segundo ejemplo

Estados



Estados

- **Compartimento del nombre** (cada estado debe tener un nombre): En el primer ejemplo tenemos el nombre de estado: **Login**, mientras que en el segundo ejemplo hay dos nombres de estado: **Impagada** y **Pagada**
- **Compartimento de la lista de variables**: El segundo compartimento es el compartimento de las variables de estado, donde los atributos (variables) pueden ser listados y asignados. Los atributos son aquellos de la clase visualizados por el diagrama de estado. En el primer ejemplo tenemos la variable de estado: **fecha-login**, a la cual se le ha asignado el valor de la fecha del día
- **Compartimento de la lista de acciones**: El tercer compartimento es el compartimento de las transiciones internas, donde se listan las actividades o las acciones internas ejecutadas en respuesta a los eventos recibidos mientras el objeto está en un estado, sin cambiar de estado. La sintaxis formal dentro de este compartimento es:
  - **nombre-evento** '(' **lista-argumentos** ')' '[' **guard-condition** ']' '/' **expresión-acción**

Estados

- Las siguientes acciones especiales tienen el mismo formato, pero representan palabras reservadas que no se pueden utilizar para nombres de eventos:
  - **entry** '/' **expresión-acción**
  - **exit** '/' **expresión-acción**
- Las acciones **entry** y **exit** no tienen argumentos, pues están implícitos en ellas. Cuando se entra al estado o se sale del estado, se ejecuta la acción atómica especificada en **expresión-acción**
- La siguiente palabra clave representa la llamada de una **actividad**:
  - **do** '/' **expresión-acción**
- La transición especial do nos sirve para especificar una actividad que se ejecuta mientras se está en un estado, por ejemplo, enviando un mensaje, esperando o calculando. Dicha actividad es la que aparece en expresión-acción

Estados & Transiciones Simples

- Por último, las transiciones internas, que son esencialmente interrupciones, se especifican teniendo en cuenta la sintaxis formal. Por ejemplo:
  - **help** / **display help**
- La transición interna help representa un evento que no causa ningún cambio de estado, pues sólo muestra, en cualquier momento, una ayuda al usuario que viene expresada en display help
- En el primer ejemplo aparecen todos los tipos de eventos especificados anteriormente, junto con las **acciones** y **actividades** que se realizan al **entrar**, **salir** o **estar** en el estado **Login**
- Una **transición simple** es una relación entre dos estados que indica que un objeto en el primer estado puede entrar al segundo estado y ejecutar ciertas operaciones, cuando un evento ocurre y si ciertas condiciones son satisfechas
- Una transición simple se representa gráficamente como una línea continua dirigida desde el estado origen (**source**) hasta el estado destino (**target**)

Transiciones Simples

- Puede venir acompañada por un texto con el siguiente formato:
  - **nombre-evento** '(' **lista-argumentos** ')' '[' **guard-condition** ']' '/' **expresión-acción** '^' **claúsula-envío**
  - **nombre-evento** y **lista-argumentos** describen el evento que da lugar a la transición y forman lo que se denomina **event-signature**
  - **guard-condition** es una condición (expresión booleana) adicional al evento y necesaria para que la transición ocurra
    - Si la **guard-condition** se combina con una **event-signature**, entonces para que la transición se dispare tienen que suceder dos cosas: **debe ocurrir el evento y la condición booleana debe ser verdadera**
  - **expresión-acción** es una expresión procedimental que se ejecuta cuando se dispara la transición
    - Es posible tener una o varias expresión-acción en una transición de estado, las cuales se delimitan con el carácter "/"
  - **claúsula-envío** es una acción adicional que se ejecuta con el cambio de estado, por ejemplo, el envío de eventos a otros paquetes o clases
    - Este tipo especial de acción tiene una sintaxis explícita para enviar un mensaje durante la transición entre dos estados. La sintaxis consiste de una expresión de destino y de un nombre de evento, separados por un punto

Transiciones Simples

- En la Figura siguiente tenemos un diagrama de estado para un ascensor, donde se combinan los estados con las transiciones simples

Transiciones Simples

- El ascensor empieza estando en el **primer piso**. Puede **subir** o **bajar**
- Si el ascensor está **parado** en un piso, ocurre un evento de tiempo rebasado después de un período de tiempo y el ascensor baja al primer piso
- Este diagrama de estado no tiene un punto de finalización (estado final)
- El evento de la transición entre los estados **EnPrimerPiso** y **Subiendo** tiene un argumento, **piso** (el tipo de este parámetro ha sido suprimido). Lo mismo sucede con los eventos de las transiciones entre **Parado** y **Subiendo** y entre **Parado** y **Bajando**
- El estado **Parado** (Idle state) asigna el valor **cero** al atributo **temporizador**, luego lo incrementa continuamente hasta que ocurra el evento **Bajar** (piso) o el evento **Subir** (piso) o hasta que la **guard-condition** [temporizador = tiempo-límite] se convierta en **verdadera**

Transiciones Simples

- La transición de estado entre Parado y EnPrimerPiso tiene una guard-condition y una expresión-acción. Cuando el atributo temporizador es equivalente a la constante tiempo-límite, se ejecuta la acción Bajar (primerpiso) y el estado del ascensor cambia de Parado a EnPrimerPiso. Esta transición de estado
  - [**temporizador** = **tiempo-límite**] / **Bajar** (**primerpiso**)
- se puede convertir en una claúsula-envío tal como:
  - [**temporizador** = **tiempo-límite**] ^ **Self.Bajar** (**primerpiso**)
  - donde la expresión destino es, en este caso, el propio objeto que se evalúa a sí mismo, y el nombre del evento es Bajar (primerpiso), evento significativo al objeto contenido en la expresión destino

Estados Avanzados

- Las características de los estados y de las transiciones, vistas en los apartados anteriores, resuelven un gran número de problemas a la hora de modelar un diagrama de estado. Sin embargo, hay otra característica de las máquinas de estado de UML, los subestados, que nos ayudan a simplificar el modelado de aquellos comportamientos complejos
- Un **estado simple** es aquel que no tiene estructura. Un estado que tiene subestados, es decir, estados anidados, se denomina **estado compuesto**. Un estado compuesto puede contener bien subestados secuenciales (disjuntos) o bien subestados concurrentes (ortogonales)
- **Subestados secuenciales**: Consideremos el problema de modelar el comportamiento de un **Cajero Automático** (CA). Hay tres estado básicos en los que podría estar este sistema: **Parado** (esperando la interacción del usuario), **Activo** (gestionando una transacción del cliente) y **Mantenimiento** (teniendo que actualizar el efectivo almacenado)

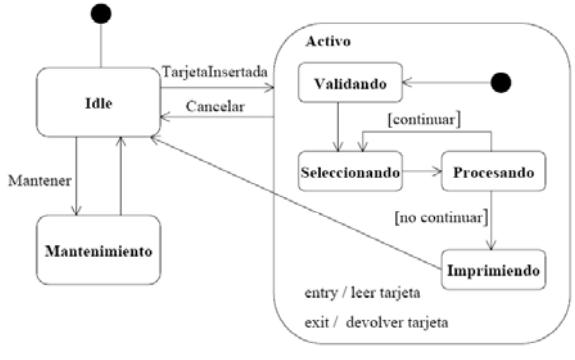
Estados Avanzados: Subestados Secuenciales

- Mientras está en **Activo**, el comportamiento de **CA** sigue un camino sencillo: validar al cliente, seleccionar una transacción, procesarla e imprimir un recibo. Después de la impresión, el CA vuelve al estado **Parado** (Idle). Podríamos representar estos estados de comportamiento como los estados **Validando**, **Seleccionando**, **Procesando** e **Imprimiendo**. Incluso sería deseable permitir al cliente seleccionar y procesar múltiples transacciones después de Validando la cuenta bancaria y antes de Imprimiendo un recibo final
- El problema que se nos plantea aquí es que, en cualquier etapa de este comportamiento, el cliente podría decidir **cancelar** la transacción, volviendo el CA al estado Idle. Usando las máquinas de estado simples, podríamos conseguir dicho efecto, pero es bastante lioso. Como el usuario podría cancelar la transacción en cualquier punto, tendríamos que incluir una transacción de forma correcta desde cada estado de la secuencia Activo. Esto es complicado porque es fácil olvidar incluir estas transacciones en todos los lugares apropiados

Estados Avanzados: Subestados Secuenciales

- Utilizando los **subestados secuenciales**, se puede resolver de una forma más sencilla este problema, tal como muestra la Figura siguiente. Aquí, el estado **Activo** tiene una subestructura, conteniendo los **subestados Validando**, **Seleccionando**, **Procesando** e **Imprimiendo**
- El estado del CA cambia de **Parado** a **Activo** cuando el cliente introduce una **tarjeta** de crédito en el cajero
- Al entrar al estado Activo, se ejecuta la acción de entrada, **entry**: leer tarjeta, que viene especificada dentro de dicho estado. Empezando con el estado inicial de la subestructura, el control pasa al estado Validando, después al estado Seleccionando y luego al estado Procesando. Después de Procesando, el control puede regresar a Seleccionando (si el usuario ha seleccionado otra transacción) o puede ir a Imprimiendo. Después de Imprimiendo, hay una transición de vuelta, sin evento ni condiciones de disparo (trigger), al estado Parado. Hay que fijarse que el estado Activo tiene una acción de salida, **exit**, la cual devuelve la tarjeta de crédito al cliente

Estados Avanzados: Subestados Secuenciales



Estados Avanzados: Subestados Secuenciales

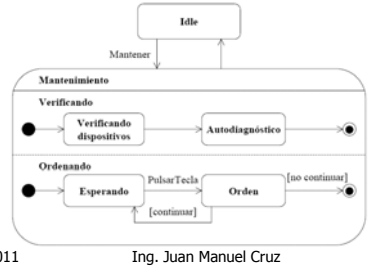
- La transición del estado **Activo** al estado **Idle** ha sido disparada por el evento **Cancelar**. En cualquier subestado de Activo, el cliente podría cancelar la transacción y que el CA volviese al estado **Parado** (pero sólo después de devolver la tarjeta de crédito al cliente, ya que la acción **exit** es enviada al dejar el estado Activo, y no importa qué causó una transición fuera de este estado).
  - **Sin subestados, necesitaríamos una transición disparada por Cancelar en cada estado de la subestructura**
- Los subestados tales como **Validando** y **Procesando** se llaman **subestados secuenciales** o **disjuntos**. Dado un conjunto de subestados disjuntos encerrados en un **estado compuesto**, se dice que el objeto está en el estado compuesto y en uno sólo de sus subestados a la vez. Por lo tanto, los subestados secuenciales dividen el espacio del estado compuesto en estados disjuntos

Estados Avanzados: Subestados Secuenciales

- Desde fuera de un estado compuesto, una transición puede tener como meta el estado compuesto o como meta un subestado
  - Si su objetivo es el **estado compuesto**, la **máquina de estado anidada** (o lo que es lo mismo, el diagrama de estado con subestados) tiene que incluir un **estado inicial**, al cual pasa el control después de entrar al estado compuesto y después de ejecutar la acción **entry** (si la hay)
  - Si su objetivo es un **subestado del estado compuesto**, el control pasa a dicho subestado, después de ejecutar la acción **entry** (si la hay) del **estado compuesto** y luego la acción **entry** (si la hay) del **subestado**
- Una transición al salir de un estado compuesto puede tener como su origen el estado compuesto o un subestado del mismo. En ambos casos, el control primero deja el estado anidado (y su acción **exit**, si la hay, es ejecutada), luego deja el estado compuesto (y su acción **exit**, si la hay, es ejecutada). Una transición cuyo origen es el estado compuesto esencialmente interrumpe la actividad de la máquina de estado anidada
  - Una máquina de estado secuencial anidada puede tener un estado inicial y un estado final

Estados Avanzados: Subestados Concurrentes

- Los subestados secuenciales son el tipo más común de máquina de estado anidada. Sin embargo, en ciertas situaciones de modelado tenemos que especificar **subestados concurrentes**. Estos subestados nos permiten especificar dos o más **máquinas de estado** que se ejecutan en **paralelo**
  - La Figura siguiente nos muestra una expansión del estado **Mantenimiento**, que aparece en la Figura anterior



Estados Avanzados: Subestados Concurrentes

- El estado de **Mantenimiento** está descompuesto en dos subestados concurrentes, **Verificando** y **Ordenando**, que se encuentran anidados en dicho estado pero separados por una **línea discontinua**. Cada uno de estos subestados concurrentes a su vez está descompuesto en subestados secuenciales
  - Cuando el control pasa del estado **Parado** al estado **Mantenimiento**, el control se **bifurca** en **dos flujos concurrentes** y el objeto implicado estará en el estado **Verificando** y también en el estado **Ordenando**. Mientras se encuentre en el estado **Ordenando**, el objeto estará bien Esperando o bien en el estado **Orden**
  - La **ejecución** de estos dos subestados concurrentes continúa en **paralelo**. Luego cada máquina de estado anidada alcanza su final. Si un subestado concurrente alcanza su final antes que el otro, el control en dicho estado espera a su estado final. Cuando ambas máquinas de estado anidadas alcanzan sus estados finales, el control de los dos subestados concurrentes **se juntan** de nuevo en un **único flujo**.
  - Una máquina de estado concurrente anidada no tiene un estado inicial y un estado final. Sin embargo, los subestados secuenciales que componen un estado concurrente pueden tener dichos estados

Diagrama de Estado: Paso a Paso

- Partir de las especificaciones del sistema
  - Primer paso — Identificar los eventos y las acciones
  - Segundo paso — Identificar los estados
  - Tercer paso — Agrupar por jerarquías
  - Cuarto paso — Agrupar por concurrencia
  - Quinto paso — Añadir las transiciones
  - Sexto paso — Añadir las sincronizaciones
- Elegida una herramienta de software: Editar, Verificar y Validar (Simular) el diagrama de estado
- Culminar con la generación del código (opción posible dependiendo de la herramienta de software)

Diagrama de Estado: Paso a Paso

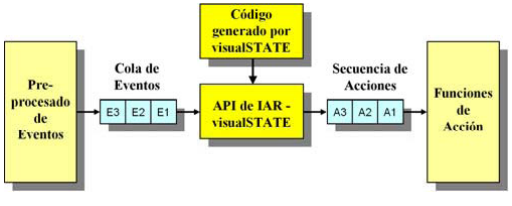
- Para las clases prácticas utilizaremos el software: **IAR visualSTATE®** (versión de evaluación limitada a 20 estados sin vencimiento), se descarga previo registro de:
  - V6.3: http://supp.iar.com/Download/SW/?item=VS-EVAL
  - Esta versión permite **editar**, **verificar** y **validar** (**simular!!!!!**) statecharts básicos pero útiles
- Para la generación del código tenemos dos opciones:
  - Utilizar el **IAR Embedded Workbench** (versión de evaluación de 30 días y kickstar) para el micro deseado
  - Generar código respetando la estructura del **pseudocódigo** y la **API** de IAR VisualSTATE

Generación de Código

- Para crear una aplicación sobre un micro dado con las herramientas de IAR **visualSTATE®** & **Embedded Workbench** se necesita código fuente de tres tipos:
  - Código generado por el usuario
  - Código generado automáticamente por visualSTATE
  - La API (Application Programming Interface) de IAR visualSTATE
- Dado que los dos últimos tipos de código los genera visualSTATE, el diseñador sólo tiene que escribir manualmente el siguiente código:
  - Código para inicializar el **hardware**
  - Código para procesar los dispositivos de **salida** (funciones de acción)
  - Código para procesar las **entradas** (generar los eventos y manejar la cola de eventos)
  - La función **main**
  - El usuario tiene que escribir a mano una parte pequeña del código

Generación de Código

- Las aplicaciones realizadas con **visualSTATE®** requieren tres tipos de código, el usuario tiene que escribir a mano una parte pequeña del mismo



- Una solución alternativa es generar código respetando la estructura del **pseudocódigo** y la **API** de IAR visualSTATE

Referencias

- Introducción al UML - Mari Carmen Otero Vidal
- G. Booch, J. Rumbaugh, I. Jacobson, "El lenguaje unificado de modelado", 2ª Edición, Addison-Wesley, 2006
- C. Larman, "UML y Patrones: Una introducción al análisis y diseño orientado a objetos y al proceso unificado", Prentice-Hall, 2003
- http://www.uml.org/
- Ingeniería de Software I, DC - FCEyN – UBA
- Introducción al uso de los Statecharts para el diseño de Sistemas Embebidos, Mariano Barrón Ruiz
- Desarrollo de Software Basado en Modelos para Sistemas Embebidos, Mariano Barrón Ruiz
- Curso de Programación de Sistemas Embebidos con Statecharts, Mariano Barrón Ruiz