SASE 2017 SIMPOSIO ARGENTINO DE SISTEMAS EMBEBIDOS

VORTEX

EMBEDDED MAKERS

El paradigma de la programación dirigida por eventos

Ing. Leandro Francucci (lf@vortexmakes.com)

9 de Agosto 2017

Objetivo

- Comprender el paradigma de la *programación dirigida por eventos o reactiva*, como una alternativa a la hora de establecer los aspectos relacionados con la concurrencia de un sistema o parte del mismo. Esto implica entender:
  - sus características,
  - sus beneficios y desventajas, comparándola con los enfoques más tradicionales,
  - su relación con la política de planificación,
  - los elementos de software y los principios que la sustentan, y
  - su relación con UML
- Presentar su aplicación en embebidos mediante el uso de los frameworks que ofrece el mercado.

Agenda

- Concurrencia y gestión de recursos
  - Sistemas de tiempo-real.
  - Políticas de planificación equitativa y basada en prioridades
  - Recursos compartidos y la serialización de su acceso mediante secciones críticas, bloqueo por semáforos y cola de mensajes. Detallando sus beneficios y consecuencias.
- Objeto activo
  - Características y estructura interna.
  - Recepción de mensajes sincrónicos y asincrónicos.
  - Beneficios y desventajas.
  - Relación con la política de planificación subyacente.
  - Modelos de comportamiento
    - Máquinas de estados y diagramas de secuencia.
- Uso de frameworks de tiempo-real como Quantum-Leap, RKH o Rhapsody para facilitar su aplicación.

Concurrencia y gestión de recursos

Sistemas de tiempo-real

Un sistema de tiempo real es aquel cuya exactitud lógica se basa tanto en la exactitud de los resultados como en su estricto tiempo de respuesta. Los hay:

Hard real-time: aquel en el que no cumplir con sus restricciones temporales puede incurrir en una falla total y catastrófica del sistema.

Soft real-time: es uno en el que se degrada el rendimiento, sin embargo incumplir las restricciones temporales no provoca una falla total del sistema

Firm real-time: es aquel en el que incumplir algunas pocas restricciones temporales no provoca una falla total, pero perder algunas más puede conducir a una falla total y catastrófica del sistema

Mitos:
Un sistema de "tiempo-real" significa un sistema "rápido" (Real-time != Fast-time)
Un sistema de "tiempo-real" implica estrictamente utilizar un RTOS
Un sistema multitarea implica estrictamente utilizar un OS/RTOS

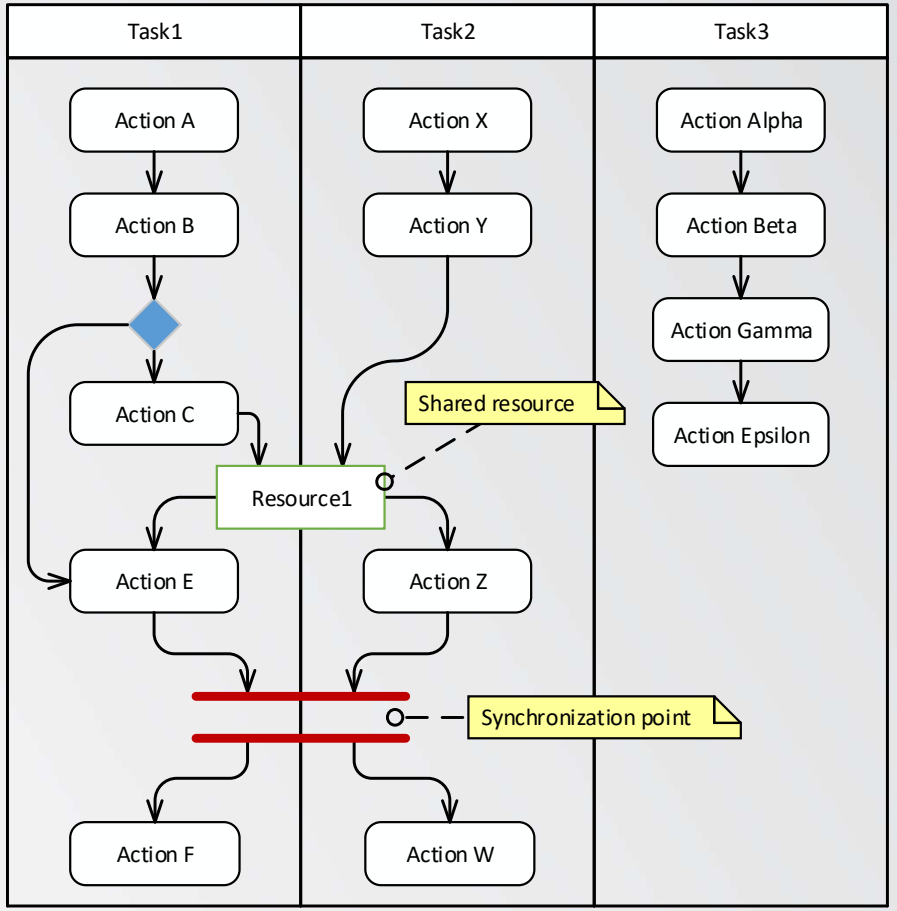
Sistemas de tiempo-real "blandos"

- Sus restricciones temporales pueden:
  a) No alcanzarse ocasionalmente,
  b) No alcanzarse por pequeñas desviaciones de tiempo, u
  c) Ocasionalmente omitirse.

Unidad de concurrencia

Concurrencia se refiere a la ejecución simultánea de secuencia de acciones

Una unidad de concurrencia (tarea o thread) tiene una secuencia de acciones en la cual el orden de ejecución es conocido, sin embargo el orden de ejecución de acciones entre diferentes unidades de concurrencia es "desconocido e irrelevante" (excepto en los puntos de sincronización)



Política de planificación
Equitativa

Política de planificación	Descripción	Pros	Cons
Cyclic Executive	The scheduler runs a set of tasks (each t completion) in a never-ending cycle. Task set is fixed at startup.	Fair, very simple, highly predictable	Unresponsive, unstable, nonoptimal, nonrobust, requires tuning, short task
Time-Triggered Cyclic Executive	Same as cyclic executive except that the start of a cyclic is begun in response to a time event so that the system pauses between cycles.	Fair, very simple, highly predictable, resynchronizes cycle with reference clock	Unresponsive, unstable, nonoptimal, nonrobust, requires tuning, short task
Round Robin	A task, once started, runs until it voluntarily relinquishes control to the scheduler. Tasks may be spawned or killed during the run.	Fair, more flexible than cyclic executive, simple	Unresponsive, unstable, nonoptimal, nonrobust, short task
Time-Division Round Robin	A round robin in which each task, if it does not relinquish control voluntarily, is interrupted within a specified time period, called a time slice.	Fair, more flexible than cyclic executive or round robin, simple, robust	Unresponsive, unstable, nonoptimal

Ejecución cíclica
Ejemplo

```
1 void
2 main( void )
3 {
4                                   /* global static and stack data */
5     static int nTasks = 3;
6     int currentTask;
7
8     currentTask = 0;                      /* initialization code */
9
10    if(POST())                    /* Power On Self Test succeeds */
11    {
12        while(TRUE)                       /* scheduling executive */
13        {
14            task1();
15            task2();
16            task3();
17        };                         /* end cyclic processing loop */
18    }
19 }
```

- Aquí, una tarea es una secuencia de código que retorna cuando esta haya terminado.
- Generalmente, una tarea que corre en un ambiente multitarea cooperativo, se ejecuta en un lazo pero contiene puntos en los cuales voluntariamente devuelve el control al scheduler para permitir que se ejecuten otras tareas.

Política de planificación
Basada en prioridad

Política de planificación	Descripción	Pros	Cons
Rate Monotonic Scheduling (RMS)	All tasks are assumed periodic, with their deadline at the end of the period. Priorities are assigned as design time so that tasks with the shortest periods have the highest priority.	Stable, optimal, robust	Unfair, may not scale up to highly complex systems
Deadline Monotonic Scheduling (DMS)	Same as RMS except it is not assumed that the deadline is at the end of the period, and priorities are assigned at design time, based on the shortness of the task's deadline.	Stable, optimal, robust	Unfair, handles tasks more flexibly
Earliest Deadline Scheduling (EDS)	Priorities are assigned at runtime when the task becomes ready to run, based on the nearness of the task deadlines—the nearer the deadline, the higher the priority.	Optimal, scales up better than RMS or DMS, robust	Unstable, unfair, lack of RTOS support
Least Laxity (LL)	Laxity is defined to be the time-to-deadline minus the remaining task-execution-time. LL scheduling assigns higher priorities to lower laxity values.	Optimal, robust	In naïve implementation causes thrashing, unstable, unfair, even less RTOS support than EDS, more complex
Maximum Urgency First (MUF)	MUF is a hybrid of LL and RMS. A critical task set is run using the highest set of priorities under an RMS schedule, and the remaining (less critical) tasks run at lower priorities, scheduled using LL	Optimal, robust	Critical task set runs preferentially to other tasks, to some stability is achieved, although not for the LL task set, in naïve implementation causes thrashing, unstable, unfair, even less RTOS support than EDS, more complex

Estructura tarea apropiativa
Ejemplo

```
24 void
25 taskA(void)
26 {
27     int stufftoDo;                  /* more private task data */
28     stuffToDo = 1;                     /* initialization code */
29
30     while(stuffToDo)
31     {
32         signal = waitOnSignal();
33         switch(signal)
34         {
35             case signal1:
36                 /* signal 1 processing here */
37                 break;
38             case signal2:
39                 /* signal 2 processing here */
40                 break;
41             case signal3:
42                 /* signal 3 processing here */
43                 break;
44             ...
45         }
46     }
47 }
```

- Aquí, la tarea permanence en reposo, a la espera de la ocurrencia de una señal, cuando la misma ocurre, la procesa y vuelve a reposo.

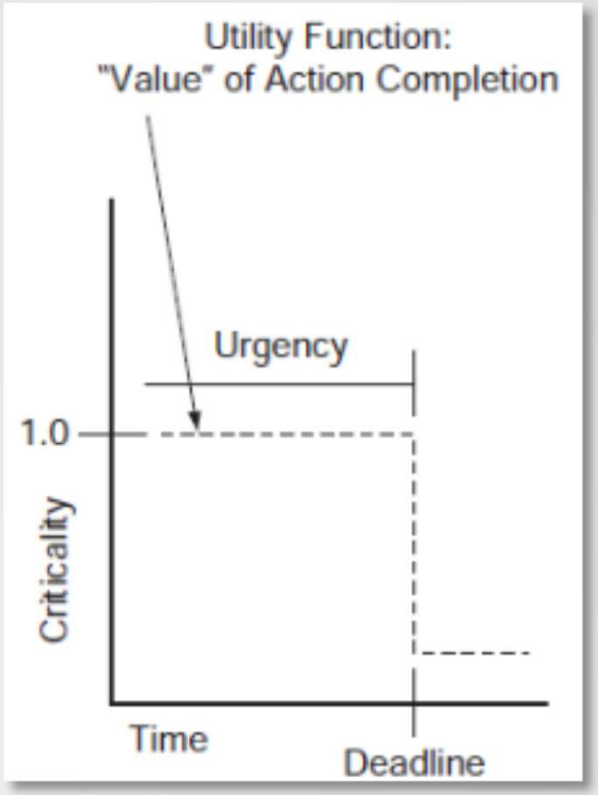
La prioridad es un valor numérico utilizado para determinar que tarea, del conjunto actual de tareas listas, se ejecutará preferencialmente

Urgencia vs. Criticidad

La **urgencia** es una medida de la rapidez con la cual un evento debe atenderse, expresada como la cercanía de un deadline

La **criticidad** se refiere a la importancia de completar una acción o secuencia de acciones en tiempo y forma

Un **deadline** es un punto en el tiempo, a partir del cual la realización de una acción se vuelve incorrecta e irrelevante



Recursos compartidos

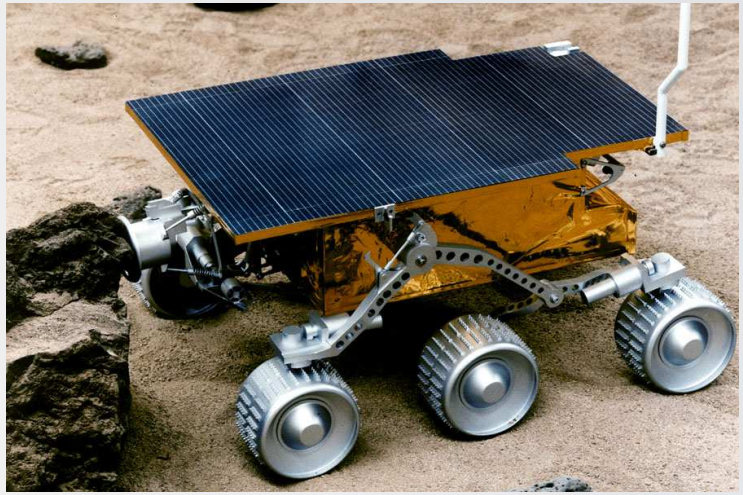
- Los sistemas de tiempo-real que ejecutan múltiples tareas, no sólo deben coordinar estas sino también el acceso a los recursos que comparten.
- El manejo de estos recursos, es crucial no sólo para la correcta operación del sistema sino también para su planificabilidad.
- Para lograr que estos puedan accederse de manera segura, por una única tarea a la vez *(denominado problema de exclusión mutua)*, lo más tradicional es serializar su acceso, mediante:
  - Secciones críticas *(Critical Region Pattern)*
  - Bloqueo por semáforos *(Guarded Call Pattern)*
  - Cola de mensajes *(Message Queueing Pattern)*

Serialización de acceso
Sección crítica

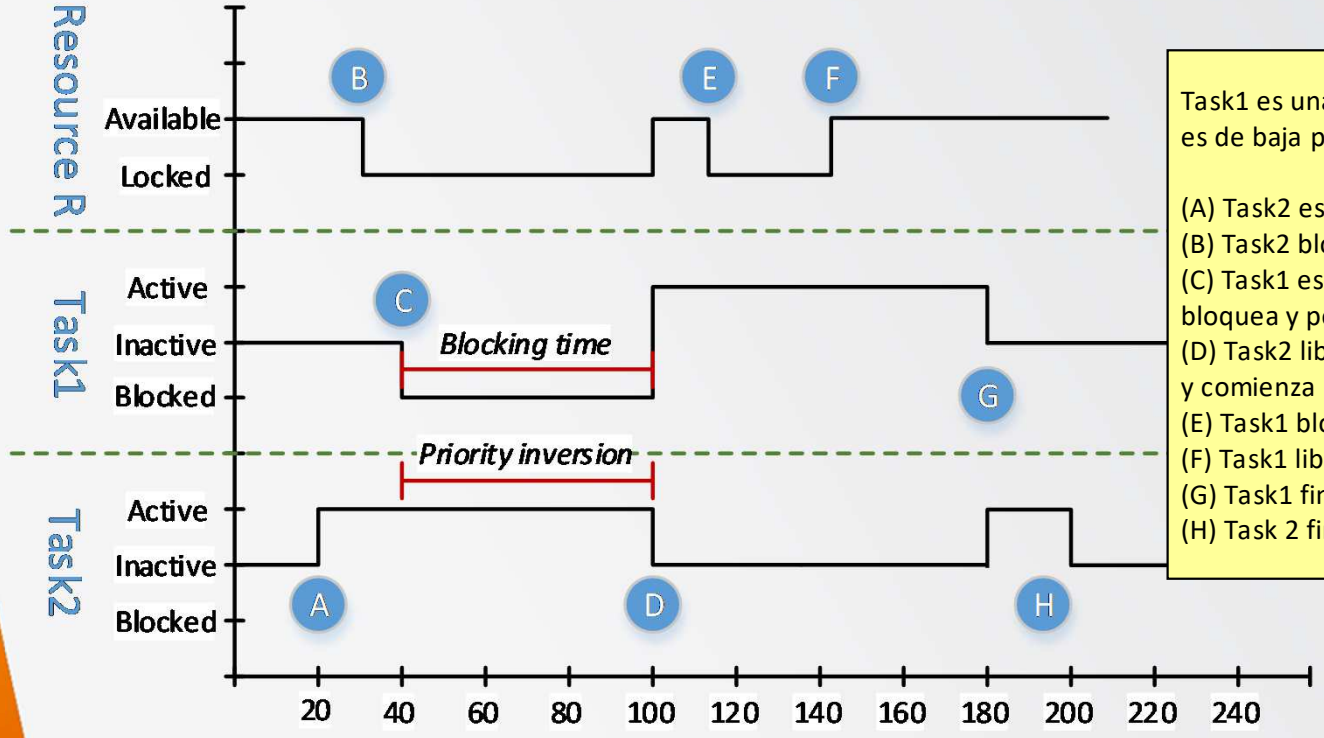
- Durante el acceso al recurso, inhibe la conmutación de tareas.
- *Lo malo:* impide que se ejecuten las tareas de alta prioridad (inversión de prioridad), aún cuando estas no acceden al recurso en cuestión, quebrando así la regla *"interrumpibilidad infinita"*
- *Lo bueno:* resuelve de manera sencilla el problema de exclusión mutua

Serialización de acceso
Bloqueo por semáforos

- Este enfoque agrega un semáforo *mutex* al recurso.
- *Lo malo:* provoca inversión de prioridad.
- *Lo peor:* su mal uso puede conducir a inversión de prioridad *ilimitada*. Ver ejemplo Mars PathFinder de Nasa.
- *Lo bueno:* resuelve el problema de exclusión mutua, permitiendo que se ejecuten aquellas tareas que no requieren el recurso.



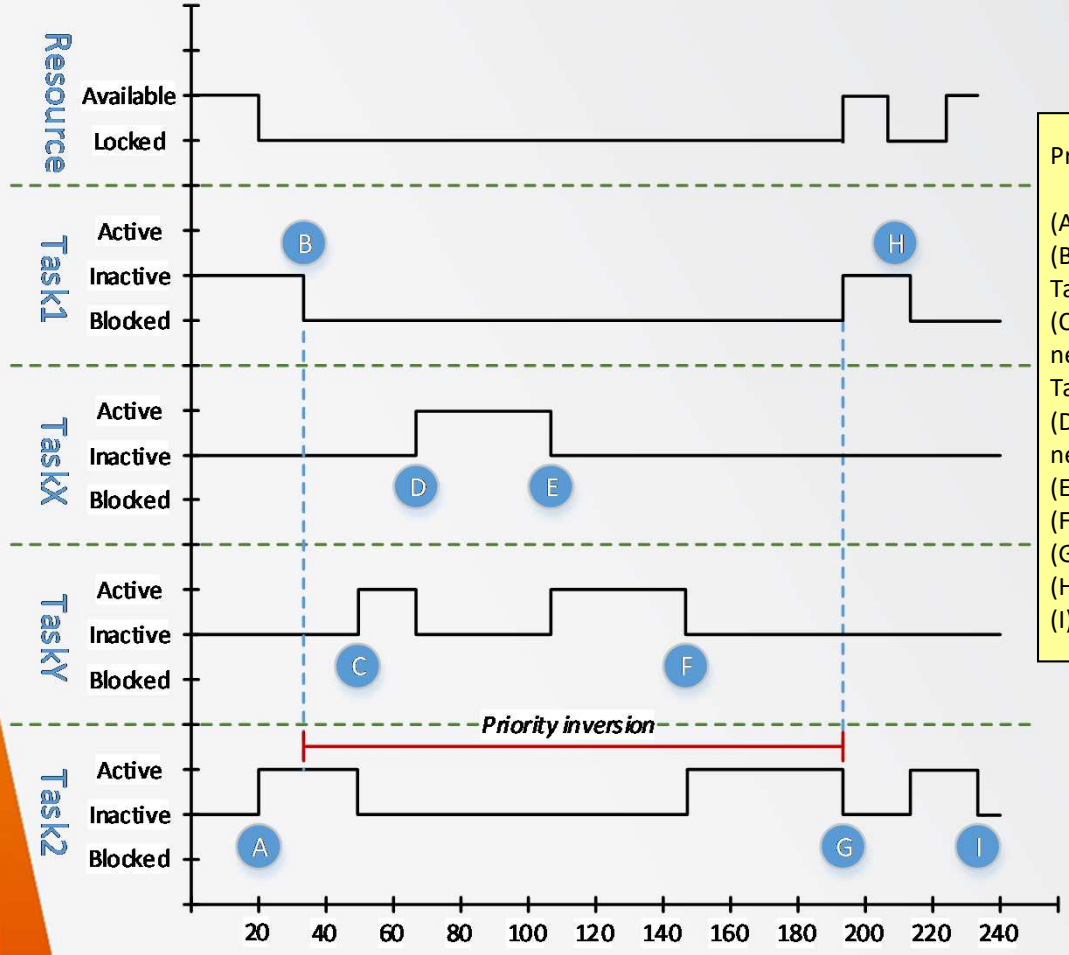
Inversión de prioridad



Task1 es una tarea de alta prioridad, mientras que Task2 es de baja prioridad.

(A) Task2 está lista y comienza su ejecución.
(B) Task2 bloquea el recurso R.
(C) Task1 está lista pero necesita el recurso, por lo que se bloquea y permite a Task2 finalizar su ejecución.
(D) Task2 libera el recurso, entonces Task1 se desbloquea y comienza su ejecución.
(E) Task1 bloquea el recurso.
(F) Task1 libera el recurso.
(G) Task1 finaliza su ejecución.
(H) Task 2 finaliza su ejecución

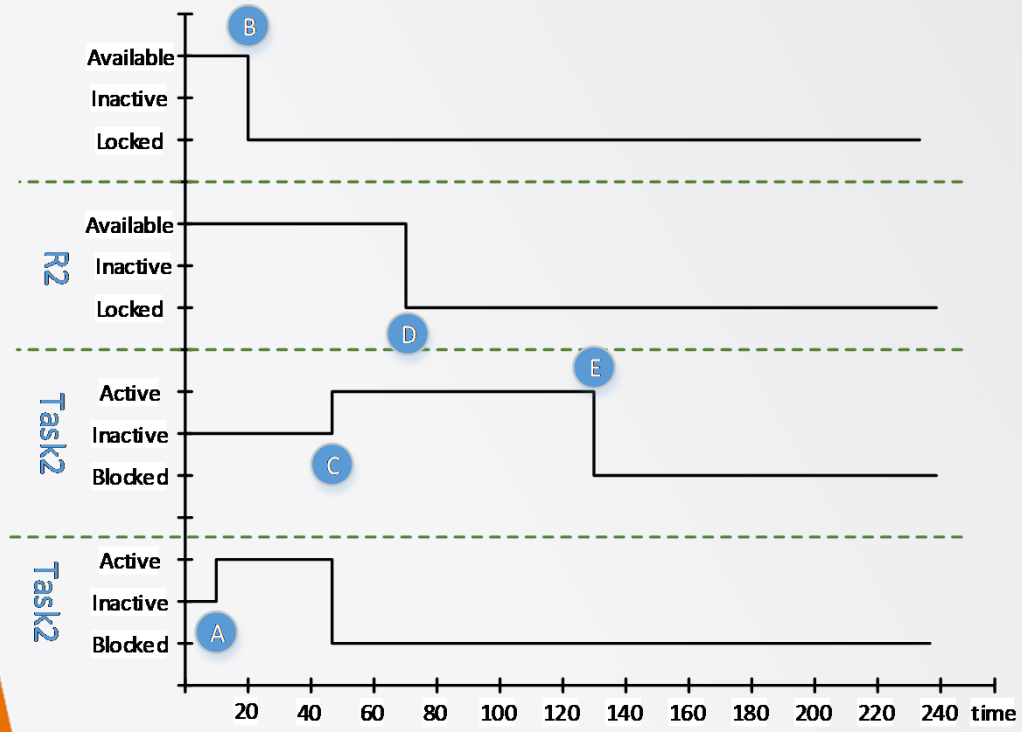
Inversión de prioridad
Ilimitada



Prioridades: Task1 > TaskX > TaskY > Task2

(A) Task2 está lista y comienza su ejecución.
(B) Task1 está lista pero necesita el recurso, por lo que se bloquea y permite a Task 2 finalizar su ejecución.
(C) TaskY, que es de mayor prioridad que Task2, esta lista. Dado que no necesita el recurso, interrumpe a Task2. Task1 queda bloqueada por Task2 y TaskY.
(D) TaskX, que es de mayor prioridad que TaskY, esta lista. Dado que no necesita el recurso, interrumpe a TaskY. Task1 queda bloqueada por 3 tareas.
(E) TaskX finaliza su ejecución, permitiendo a TaskY reanudar.
(F) TaskY finaliza su ejecución, permitiendo a Task2 reanudar.
(G) Task2 libera el recurso, permitiendo a Task1 acceder al recurso.
(H) Task1 libera el recurso y finaliza su ejecución.
(I) Task2 continua y termina su ejecución.

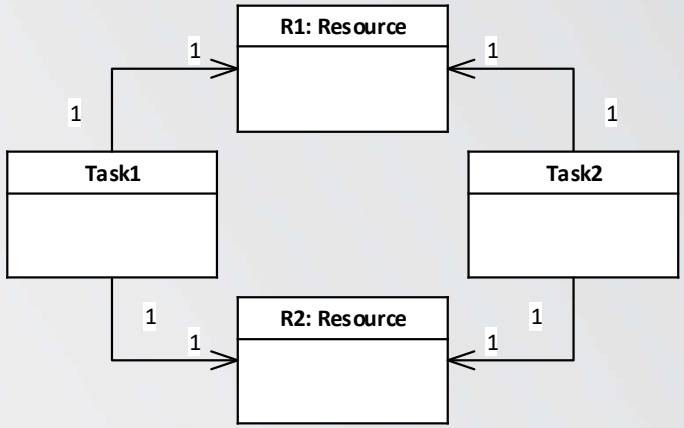
Interbloqueo



Prioridades: Task1 > Task2

(A) Task2 comienza su ejecución e intenta bloquear R1, y luego R2.
(B) Task2 bloquea R1 y está a punto de bloquear R2, cuando ...
(C) Task1 entra en ejecución, interrumpiendo Task2, con la intensión de bloquear R2, y luego R1.
(D) Task2 bloquea R2.
(E) Ahora Task2 necesita acceder a R1, pero R1 ya esta bloqueado. Task2 debe bloquearse hasta que Task1 pueda liberar R1. Sin embargo, Task1 no puede entrar en ejecución para liberar R1 ya que necesita R2, que está bloqueado por Task2.

El interbloqueo ocurre cuando un cliente de un recurso espera una condición que puede que nunca ocurra



Interbloqueo
Condiciones

- Todos estas condiciones deben cumplirse para que ocurra:
  1) Exclusión mutua (bloqueo) de recursos
  2) Los recursos están retenidos (bloqueados) mientras se espera por otros
  3) Se permite la apropiación (preemption) mientras se retienen recursos
  4) Una condición de espera circular existe

Se evita si al menos una de estas condiciones no ocurre

Serialización de acceso
Cola de mensajes

- Enfoque asincrónico, serializa las solicitudes de acceso al recurso mediante una cola, procesándolos uno por vez.
- *Lo malo:* los retardos de acceso al recurso, provocados por el encolamiento de solicitudes, pueden provocar que el tiempo de respuesta no sea el requerido.
- *Lo bueno:* resuelve el problema de exclusión mutua, permitiendo que se ejecuten las tareas de acuerdo a sus prioridades.
- *Lo mejor:* está exento de la inversión de prioridad, y el interbloqueo, provocados por los mecanismos de exclusión mutua más tradicionales, como los semáforos.

Sistema reactivo
Ejemplo

- Requerimos resolver por software la administración de un panel de control, minimizando su consumo de energía, cuyas entradas son llaves selectoras, botones, sensores, temporizadores, mensajes de otros módulos, entre otras.
- Su comportamiento está en función de sus diferentes modos de operación y del estado de sus entradas.
- La tarea que la resuelve podría no ser la única en el sistema.

Los **sistemas reactivos** reaccionan a estímulos internos y externos (eventos), efectuando la acción apropiada en un contexto particular.

Soluciones propuestas

- Monitorear continuamente las entradas, buscando cambios, los cuales provocan la ejecución de un procesamiento particular.

```
ControlPanelMgr()
{
  …
  forever
  {
    if (isChangedButtonA())
      processA(buttonAStatus);
    …
    if (isActivatedAbortSignal())
      processB(…);
    …
    if (isReceivedMessage())
      processX(…);
    …
    processUpdate();
  }
}
```

- ¿Qué ocurre?
  - si alguna entrada cambia más de una vez durante la ejecución del ciclo
  - si algunos procesos son dependientes de otros
  - si un proceso depende de más de una entrada
  - el tiempo de un ciclo en su peor caso es mayor al tiempo de cambio mínimo de una entrada
  - si las entradas tienen diferente importancia en su atención
- ¿Estamos ahorrando energía?
- ¿Cómo gestiono los modos del sistema?

Soluciones propuestas

- Convierto `ControlPanelMgr` en una tarea periódica.

```
ControlPanelMgr()
{
  …
  forever
  {
    if (syncTime)
    {
      if (isChangedButtonA())
        processA(buttonAStatus);
      …
    }
    else
      savePower();
  }
}
```

- El período se elige de acuerdo al tiempo mínimo de cambio de la entrada más rápida.
- ¿Qué ocurre si una única señal es mucho más rápida que el resto?
- ¿Qué ocurre si por algún motivo el sistema permanece durante largos períodos de tiempo sin cambios en sus entradas?

Soluciones propuestas

- Bloqueo `ControlPanelMgr` por el cambio de todas las entradas.

```
ControlPanelMgr()
{
  …
  forever
  {
    waitForSignal();
    if (isChangedButtonA())
        processA(buttonAStatus);
    …
    processUpdate();
  }
}
```

- Otras tareas independientes de `ControlPanelMgr` monitorean las entradas según su tipo, señalizándola cuando detecta un cambio.
- Esta señalización desbloquea `ControlPanelMgr` para que determine cuál(es) entrada(s) han cambiado.
- Durante el bloqueo el sistema está en reposo ahorrando energía.
- ¿Cómo resuelvo las tareas independientes en ambientes RTOS y bare-metal?

Soluciones propuestas

- Convierto a `ControlPanelMgr` en un Objeto Activo.

```
ControlPanelMgr()
{
  …
  forever
  {
    event = waitForEvent();
    process(event);
  }
}
```

- `ControlPanelMgr` tiene su propio pool (cola) de eventos que permite almacenarlos aún estando en ejecución.
- `ControlPanelMgr` permanece en reposo (bloqueado) hasta que al menos un evento se almacene en el pool.
- Respecto de los eventos:
  - generalmente se agrupan por tipo, e incluyen parámetros para su mejor administración.
  - provienen de ISR u otras tareas independientes.
  - se procesan de a uno por vez (*Run-To-Completion*), en el orden almacenados.
  - este puede ser FIFO, LIFO o por prioridad.
  - De acuerdo a la funcionalidad del sistema, su procesamiento puede representarse por máquinas de estados.

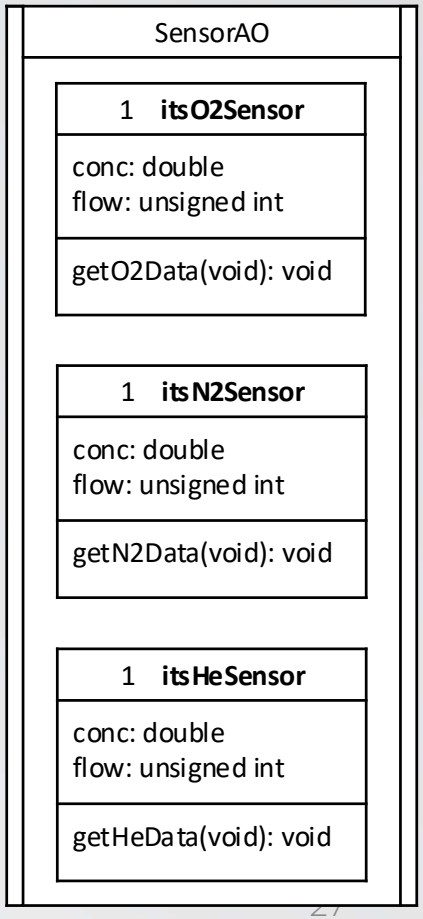
Objeto activo

Unidad de concurrencia de UML

Objeto activo
Definición y características

- El objeto activo (AO) es uno que se ejecuta en su propio hilo de ejecución (contexto). Generalmente, es un objeto estructurado, que se compone por objetos que se ejecutan en el contexto del hilo del AO que los contiene.

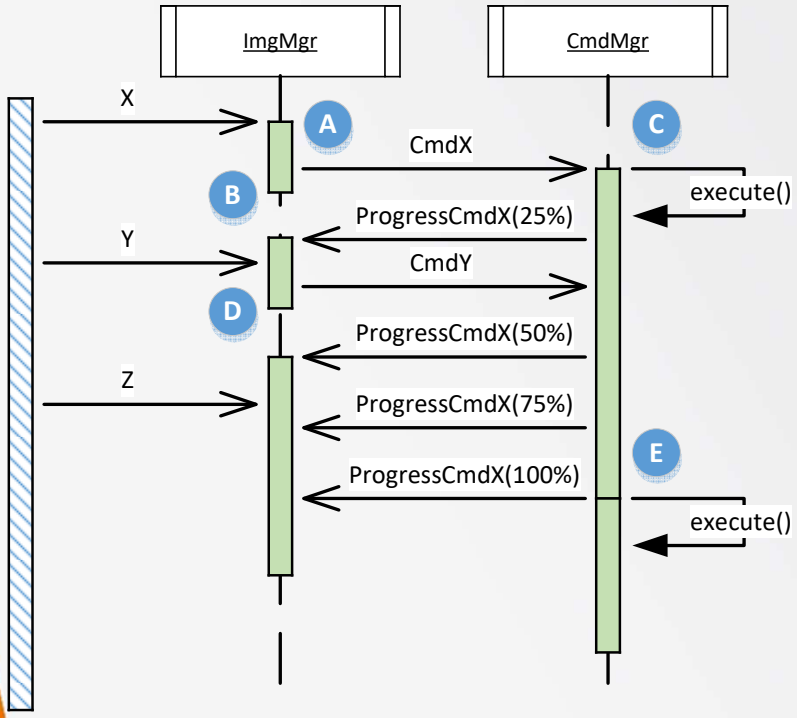
  - Tiene la responsabilidad general de
    - coordinar la ejecución interna despachando mensajes a las partes constituyentes y
    - proveer la información necesaria al OS subyacente, para que este último pueda planificar la ejecución del hilo.
  - Puede recibir mensajes asincrónicos y/o sincrónicos.
  - Posee sus propio datos y estado, su comportamiento puede o no representarse mediante una o más máquinas de estados (SM)
  - Despacha los mensajes asincrónicos de a uno por vez, cumpliendo con el comportamiento Run-To-Completion (RTC)



Objeto activo
Beneficios

- Produce un fuerte encapsulamiento y desacoplamiento.
- Suele poseer una única (o muy acotadas) responsabilidades.
- Si recibe sólo mensajes asincrónicos:
  - evita el problema de la exclusión mutua al compartir recursos en un ambiente concurrente.
  - Evita lidiar con los mecanismos de bajo nivel de los OS/RTOS, para la sincronización de tareas y serialización de acceso a recursos.
  - Respeta el procesamiento de eventos RTC, obligatorio en una máquina de estados.
  - Su codificación aplica técnicas puramente secuenciales, sin preocuparse de semáforos, y otros mecanismos de serialización. *En realidad, los oculta.*
- Son modelos de alto nivel que pueden utilizar los servicios del RTOS subyacente, proveyendo una abstracción adecuada para trabajar con la modelización propuesta por UML
- Su implementación es muy simple, cualquier OS/RTOS lo soporta.
- En general, logra el mismo nivel de respuesta de eventos, determinismo y buena utilización de CPU, que las soluciones tradicionales basadas en OS/RTOS “desnudos”.

Objeto activo
Estrategia



`ImgMgr` y `CmdMgr` son AO

A. `ImgMgr` sale de reposo debido al mensaje (asincrónico) X.

B. Luego lo despacha y procesa, enviando el mensaje `CmdX` a `CmdMgr`. No obstante continúa su ejecución, e inclusive espera la respuesta a `CmdX`. **Desacoplamiento.**

C. `CmdMgr` recibe `CmdX`, sale de reposo, despacha y procesa `CmdX`.

D. `ImgMgr` aún espera la respuesta a `CmdX` no obstante despacha Y enviando `CmdY` a `CmdMgr`.

E. Finalizado el procesamiento de `CmdX`, `CmdMgr` despacha `CmdY` para procesarlo.

Objeto activo
Mensajes asincrónicos y sincrónicos

- Los mensajes asincrónicos se despachan desde su cola de eventos interna y se procesan de a uno por vez, respetando así el comportamiento RTC.
- El orden de los eventos almacenados, y por ende su posterior procesamiento, puede ser FIFO, LIFO, por prioridad u otro criterio.
- Si durante el despacho o procesamiento de un evento asincrónico, es interrumpido por una tarea de mayor prioridad y esta envía en mensaje al AO recientemente desalojado, observar que, si el mensaje es:
  - Asincrónico, simplemente se encola.
  - Sincrónico, se procesa inmediatamente en el contexto del AO en ejecución. *Aquí no sólo es importante mantener la integridad de los datos internos, sino también la coherencia del comportamiento del AO en recepción de mensajes sincrónicos y asincrónicos.*

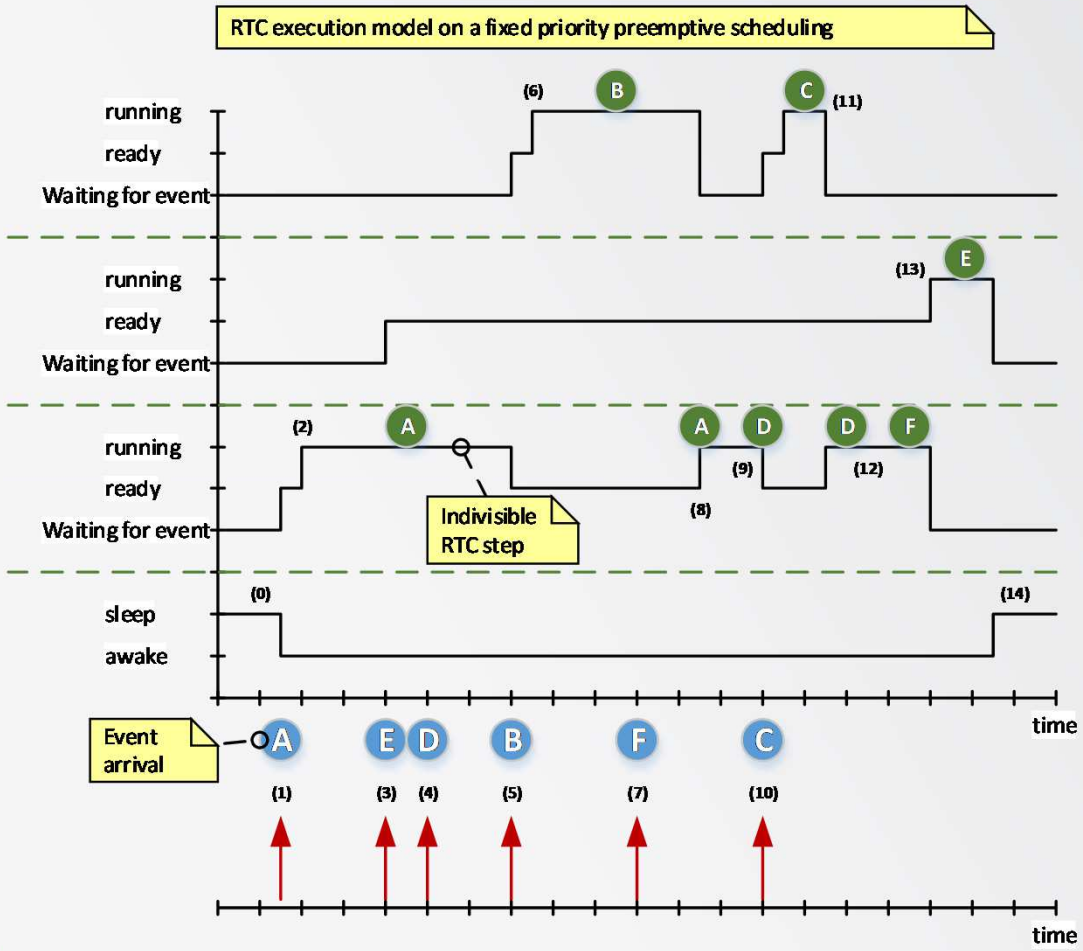
Objeto activo
Preemptive

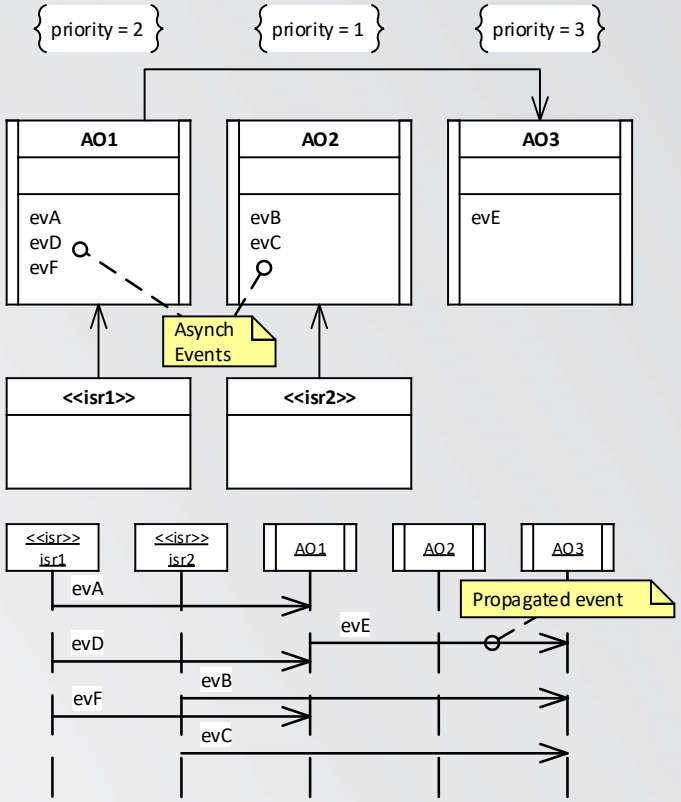
- Los objetos activos se mapean a las tareas de un RTOS/OS preemptive tradicional.
- En esta configuración el modelo se beneficia de las capacidades del RTOS/OS subyacente.

Lazo de eventos asincrónicos

```
1  thread_loop
2  {
3      running = 1;
4      while(running)
5      {
6          e = get the event from the active object's queue;
7          dispatch the event 'e' to the active object's processor;
8          recycle event 'e';
9      }
10     remove active object from the framework;
11     delete thread;
12 }
```

En ejecución





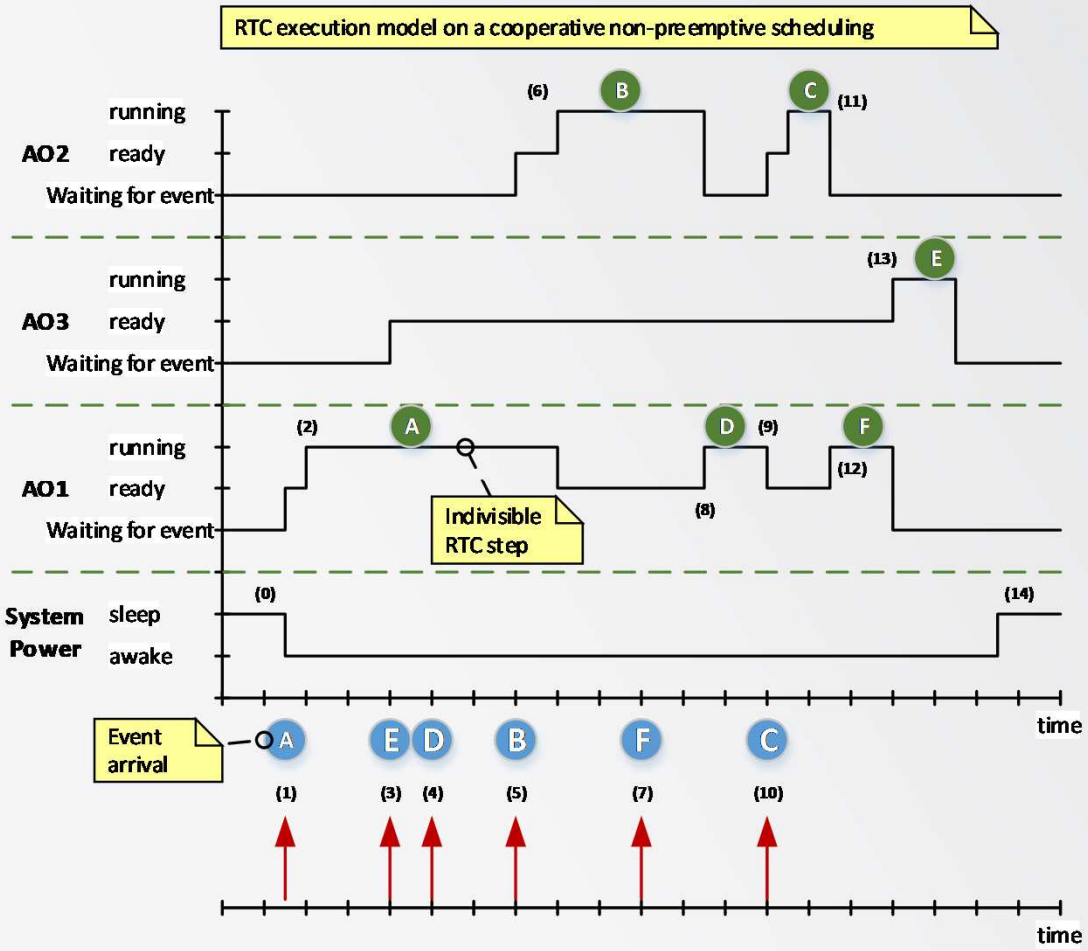
Objeto activo
Cooperative non-preemptive

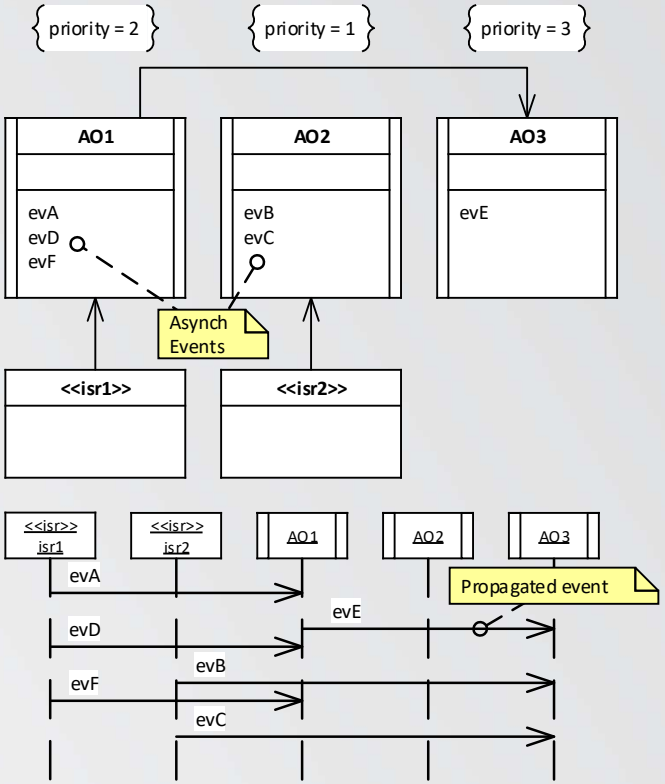
Puede implementarse mediante un lazo de eventos con política cooperative non-preemptive, que gestiona la ejecución de todos los objetos activos del sistema.

Lazo de eventos asincrónicos

```
1  infinite_loop
2  {
3      disable interrupts;
4      if (is active object ready to run)
5      {
6          find the active object with highest priority;
7          enable interrupts;
8          e = get the event from the active object's queue;
9          dispatche the event 'e' to the active object's processor;
10         recycle event 'e';
11     }
12     else
13         executes the idle processing;
14 }
```

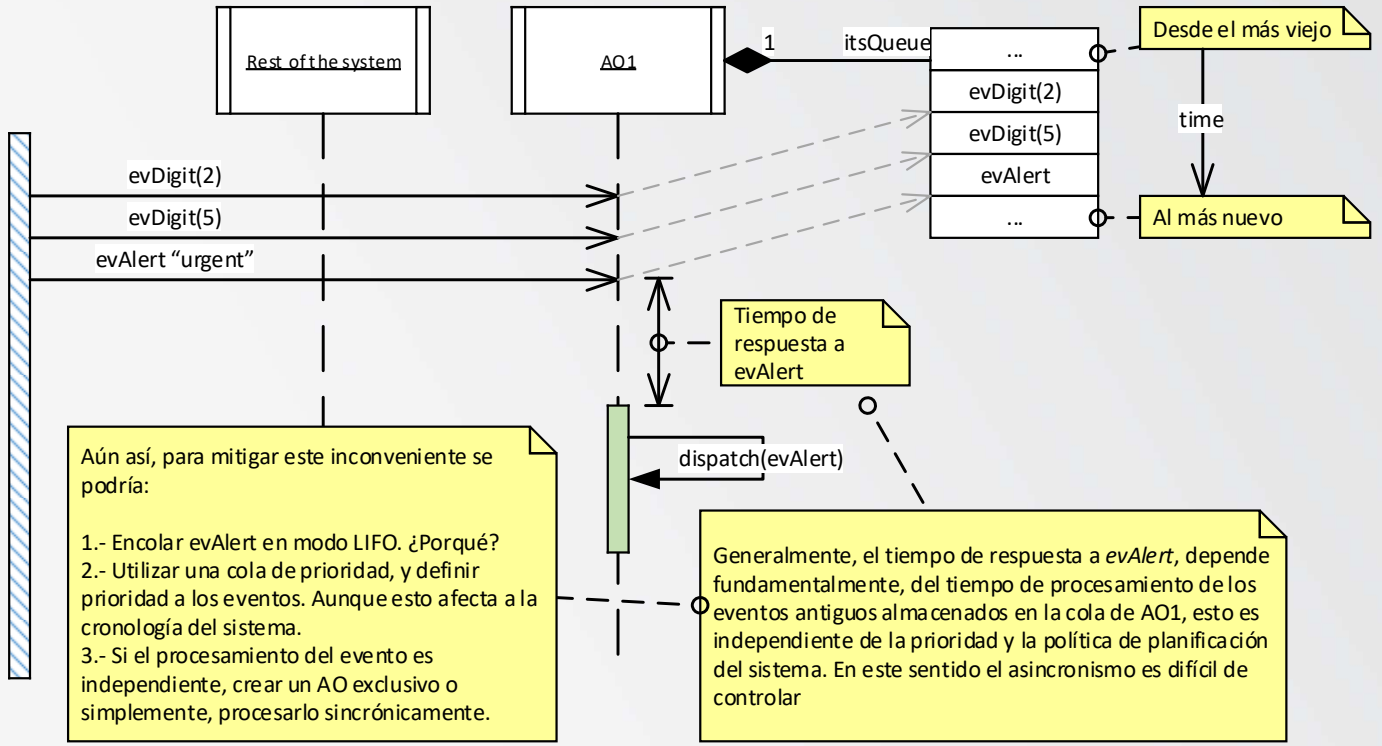
En ejecución



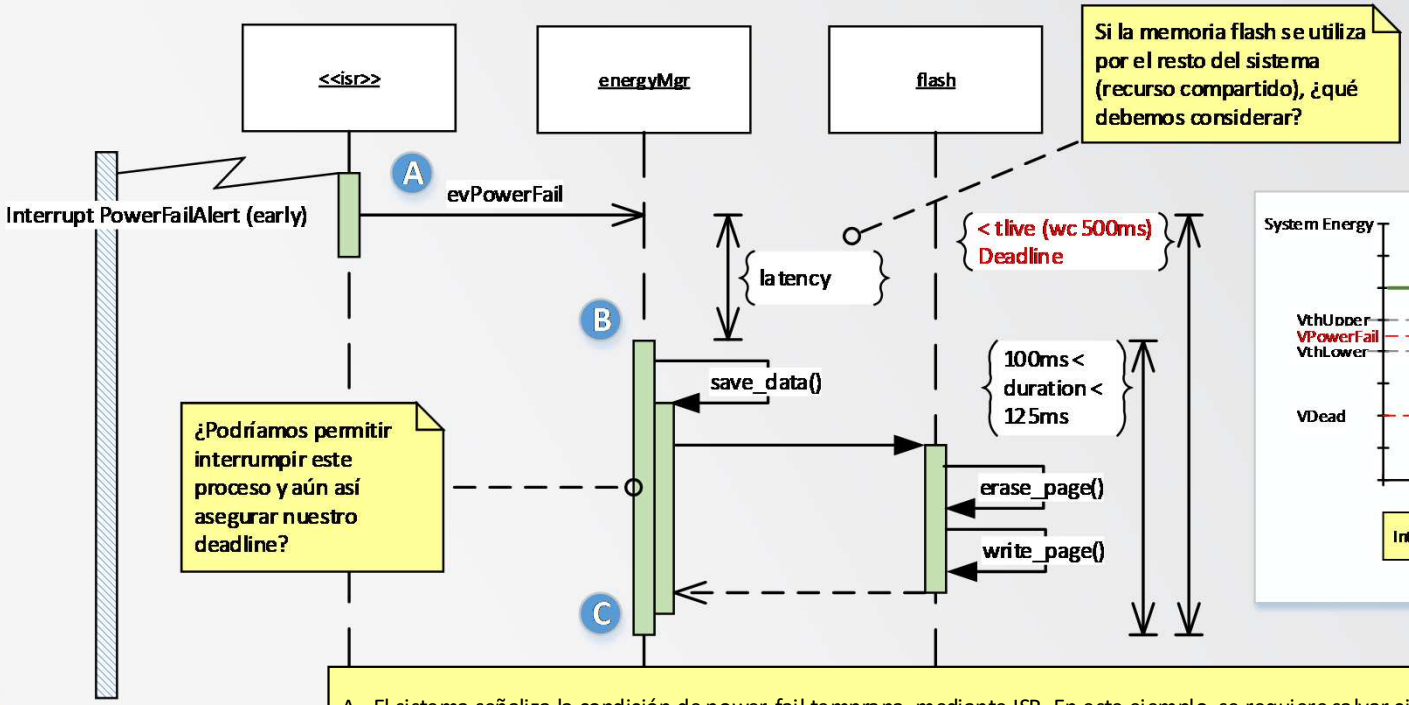


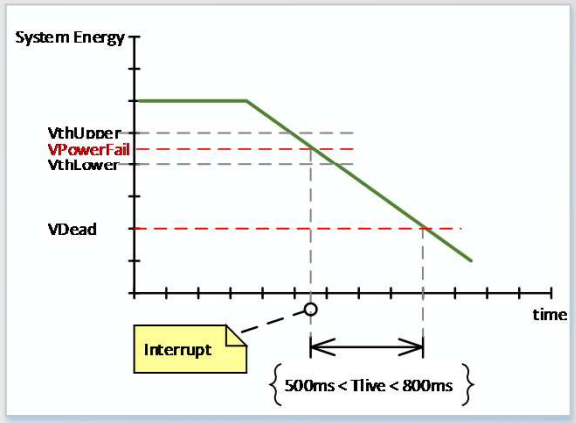
Objeto activo
Respuesta no-inmediata

- Su principal DESVENTAJA se debe justamente al intercambio de mensajes asincrónicos, *"send and forget"*, ya que el AO receptor no procesa el mensaje entrante hasta no terminar de procesar los mensajes anteriores.



Objeto activo
Eventos asincrónicos urgentes





A - El sistema señaliza la condición de power-fail temprana, mediante ISR. En este ejemplo, se requiere salvar ciertos datos en memoria no-volátil. El sistema valora esta acción como importante y de no lograrla el sistema presenta una falla grave.

La forma y tipo de señalización depende del sistema, entre muchas alternativas:

A1 – Salvar los datos directamente desde la misma interrupción, llamada a función (sincrónica) en contexto de la ISR.
A2 – Enviar un mensaje asincrónico a un AO para que lo procese.
A3 – Desde la ISR únicamente señalizar al sistema con algún mecanismo disponible (event-flag, semaphore, signal u otro). Luego en contexto de AO, procesar el evento.

En todos los casos bosquejar las alternativas propuestas y si es posible detallar alguna otra.

B – El sistema responde al evento, y comienza su procesamiento.
C – Finalmente, los datos se salvan en memoria, *siempre y cuando no hayamos alcanzado el deadline tlive*.

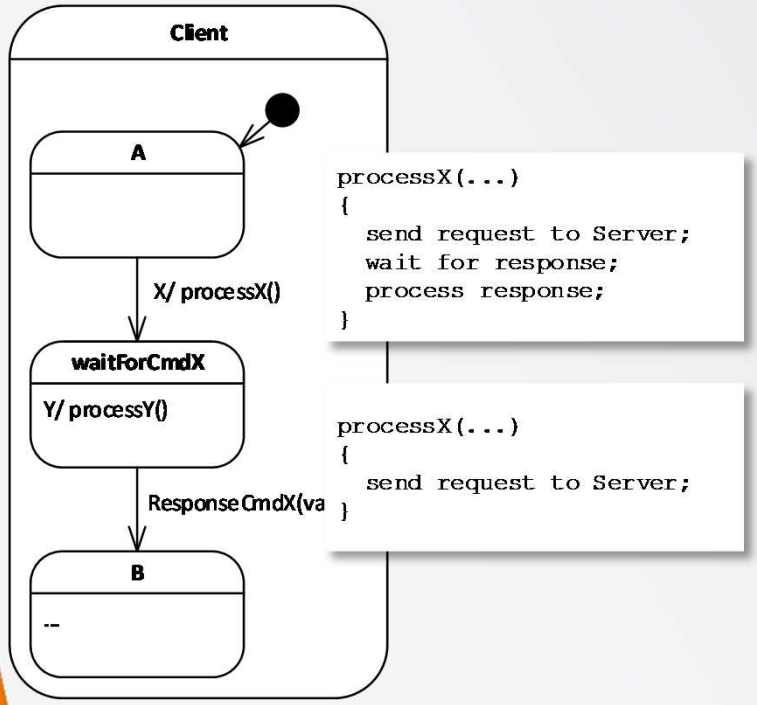
Mal uso de Objetos Activos

- Supongamos un AO `Client` que entre otras cuestiones, envía mensajes a otro, `Server`, para que los procese y devuelva un resultado.

```
Client() {
  …
  forever {
    msg = waitForAMessage();
    switch (msg.id) {
      case MSG_ID_X:
        send request X to Server AO;
        wait for response (*)
        process response
        break;
      case MSG_ID_Y:
        processY(msg);
        break;
      ...
      default:
        invalidMsg(msg);
        break;
    }
  }
}
```

- **(*)** `Client` se bloquea esperando la respuesta de `Server`. ¿Qué ocurre si en esta condición:
- ciertos mensajes enviados a `Client` requieren respuesta dentro de un plazo determinado (deadline)?
- `Client` requiere responder a un ping de un software watchdog timer?
- `Client` y `Server` comparten un recurso?

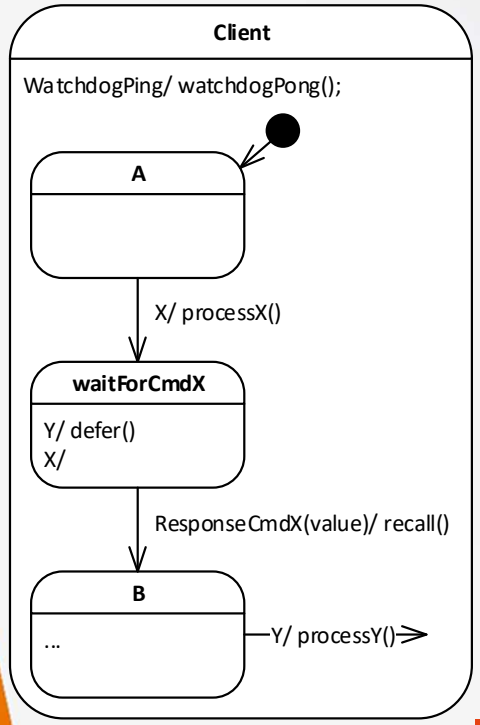
Estado de Objetos Activos



- Para que en la condición (*), `Client` continúe respondiendo mensajes, este debe tener memoria de dicha situación e inmediatamente volver al lazo de eventos `forever{}`.
- Aún así, ¿qué ocurre si en `waitForCmdX` recibe Y?, en estas circunstancias, ¿es posible enviar un nuevo mensaje a `Server`?

- ***No es posible***:
  - Ignoramos Y
  - Diferimos Y hasta recibir la respuesta de Server
- ***Si es posible***:
  - ¿Qué ocurre si Server responde al segundo mensaje antes que al primero?
  - ¿Cuál es el límite permitido de mensajes a Server cuando está ocupado?
  - ¿Qué ocurre si hay otros mensajes a otro AO, que dependen de los enviados a Server?

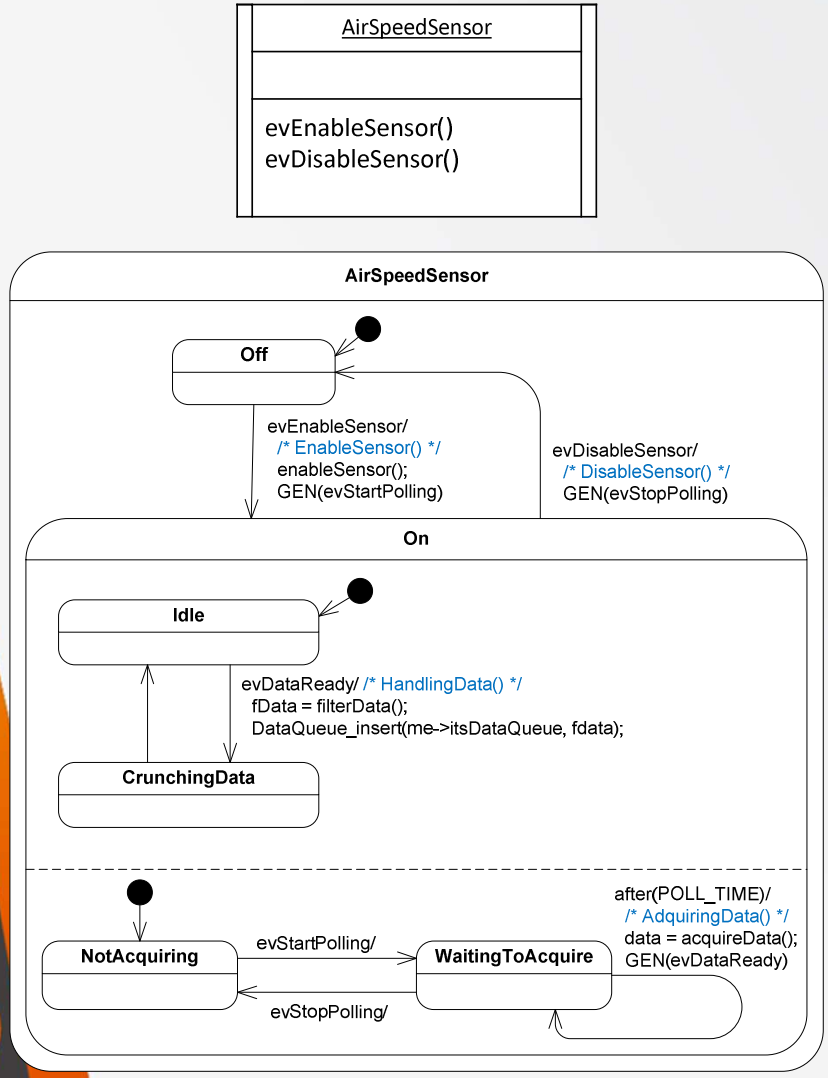
Estado de Objetos Activos



- Para resolver estas y otras condiciones más complejas, con la estrategia de concurrencia propuesta por los Objetos Activos, es preferible trabajar con el enfoque basado en estados.
- Que respecto del enfoque tradicional (in-line blocking):
  - Los bugs son más fáciles de encontrar y arreglar
  - Las soluciones son más flexibles, seguras y fiables.

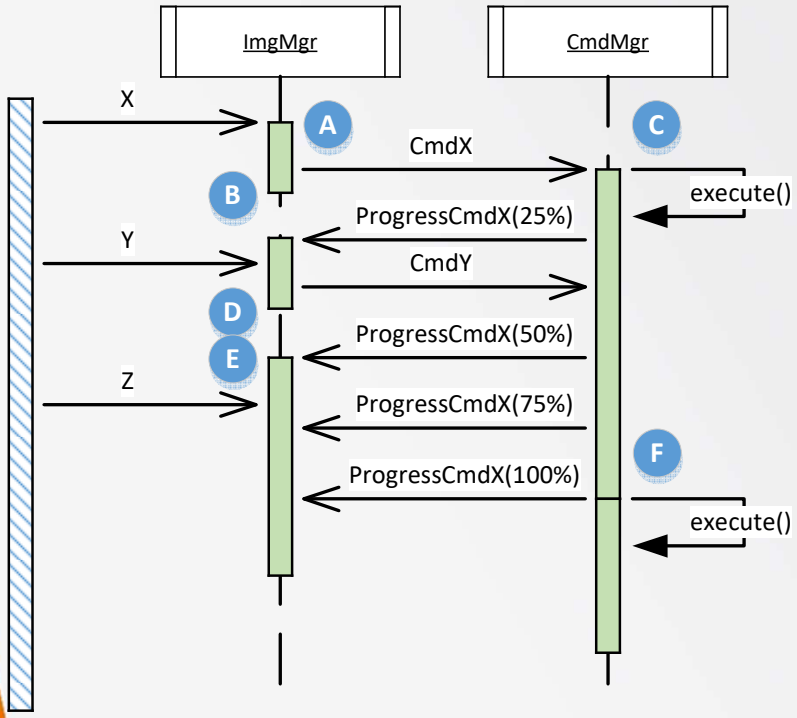
Logrando así un sistema **más estable**, **robusto**, **fácil de mantener** y **extensible**, que además exhibirá una performance más predecible.

Objeto activo reactivo
Con máquinas de estados



- El comportamiento de un AO reactivo suele representarse mediante *una o más* SMs.
- Sabiendo que:
  - El sistema modelado posee un número finito de condiciones de existencia, llamados estados.
  - El comportamiento dentro de un estado dado se define por:
    - Los eventos aceptados
    - Las acciones asociadas al entrar y salir del estado
    - Las actividades efectuadas mientras permanece en el estado
    - El conjunto completo de pares “transición-estado destino”
  - El sistema reside en sus estados por un período de tiempo significativo
  - El sistema puede cambiar de estado mediante un número finito de maneras bien definidas, llamadas transiciones.
  - Las transiciones se ejecutan hasta finalizar, incluyendo la ejecución de las acciones involucradas.

Objeto activo reactivo
Sin máquinas de estados



El comportamiento de un AO reactivo no es *estrictamente* necesario representarlo mediante SM.

El AO `CmdMgr` despacha mensajes y los procesa en su propio contexto de ejecución respetando el comportamiento RTC.

A. `ImgMgr` sale de reposo debido al mensaje X.

B. Luego lo despacha y procesa, enviando el mensaje `CmdX` a `CmdMgr`. No obstante continúa su ejecución, e inclusive espera la respuesta a `CmdX`. **Desacoplamiento.**

C. `CmdMgr` recibe `CmdX`, sale de reposo, despacha y procesa `CmdX`.

D. `ImgMgr` procesa la respuesta parcial a CmdX, al mismo tiempo recibe Y, lo cual provoca enviar `CmdY` a `CmdMgr`, el cual lo almacena.

E. Mientras ImgMgr procesa ProcessCmdX recibe Z.

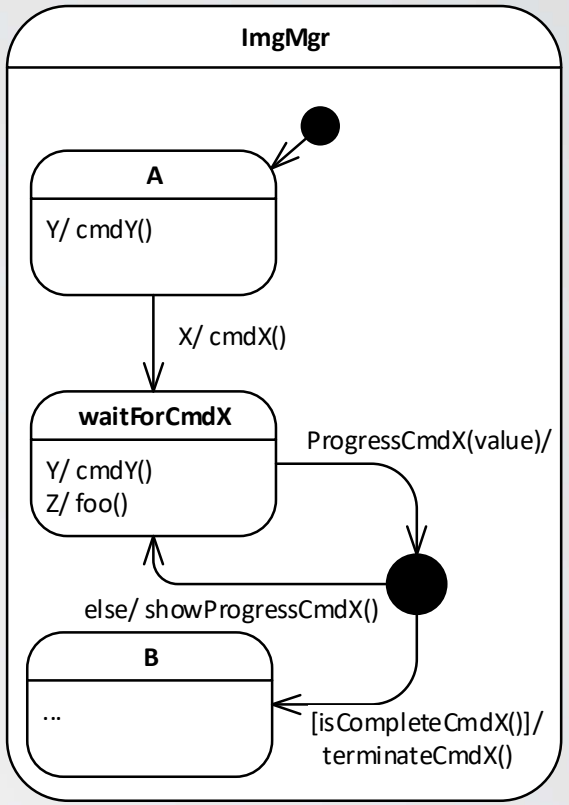
F. Finalizado el procesamiento de `CmdX`, `CmdMgr` despacha `CmdY` para procesarlo.

Objeto activo reactivo
Sin máquinas de estados

```
/* cmdx.c */
static void
execute(CmdMgr *const me)
{
  ...
  f = ((CmdX *)me)->foo;
  ...
}

/* cmdmgr.c */
static void
processCmd(CmdMgr *const me)
{
  (*me->execute)(me);
}
```

- El AO `CmdMgr` despacha mensajes y los procesa en su propio contexto de ejecución mediante la función polimórfica `processCmd()`, respetando RTC.

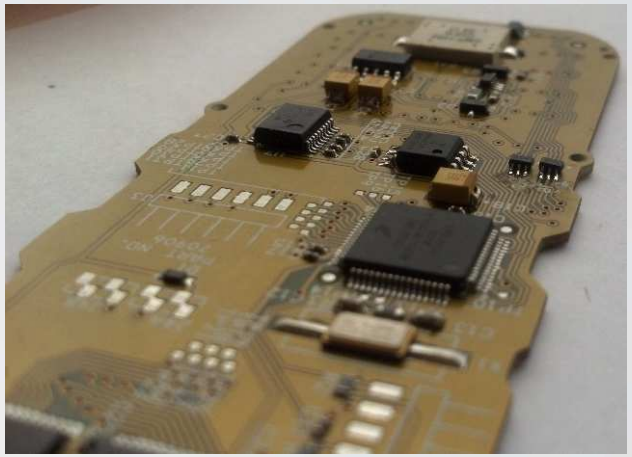


- El AO `ImgMgr`, cliente de `CmdMgr`, se representa mediante una SM.
- El estado `waitForCmdX` espera que el procesamiento de `CmdX` termine. ¿Esto perdió la ventaja del *send and forget*?
- Ocurre lo contrario si el procesamiento de `CmdX` es sincrónico, en cuyo caso `ImgMgr` se bloquea y no puede responder a ningún otro mensaje.

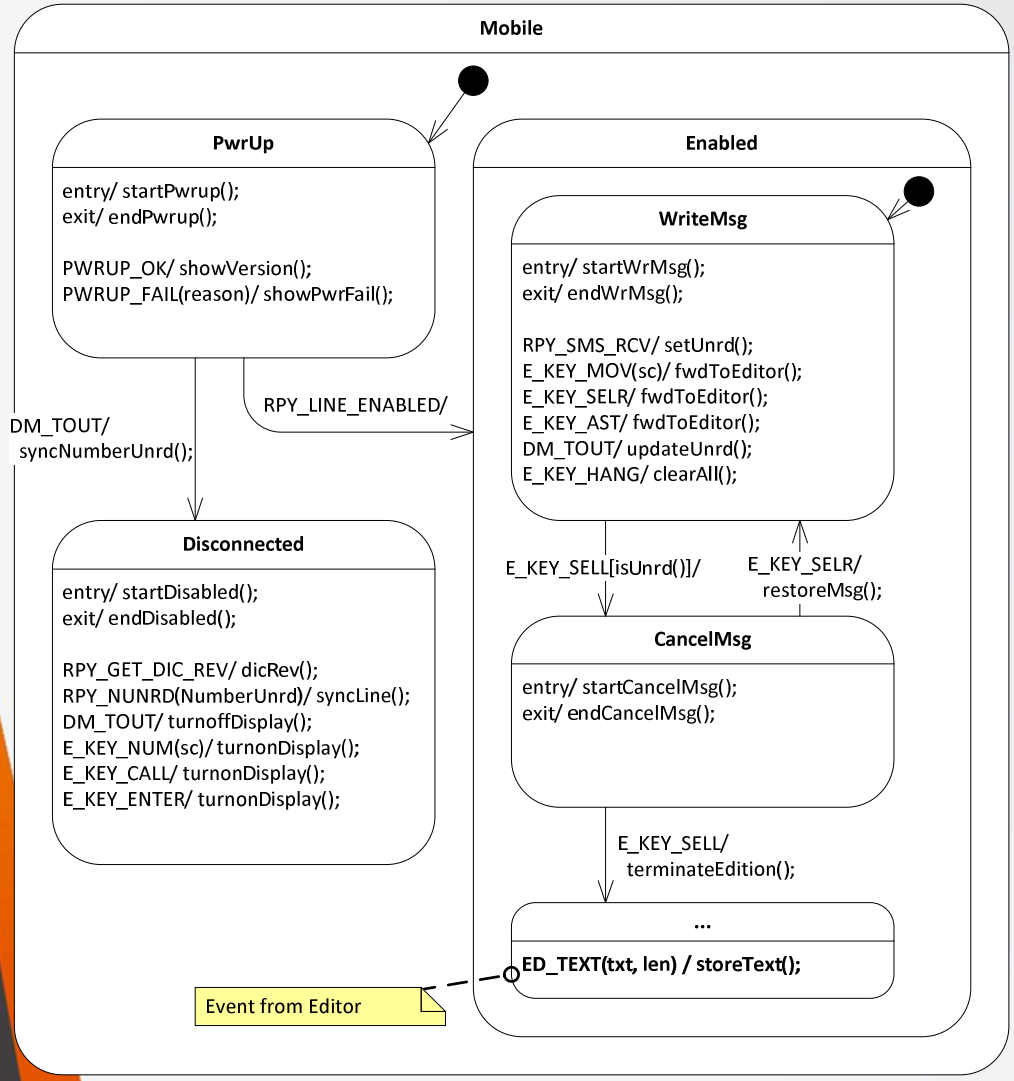
Objeto activo
Ejemplo

- Supongamos un sistema que permite, entre otras cosas, editar texto en pantalla.
- El sistema ha sido resuelto utilizando diferentes objetos activos que colaboran entre si para lograr la funcionalidad requerida.

- El objetivo aquí es mostrar y comparar diferentes soluciones para resolver la edición de texto. Específicamente, resolviendo el editor como un AO, como una región ortogonal, o como un objeto pasivo.
- El editor posee diferentes modos de operación y recibe la entrada a partir de un teclado.

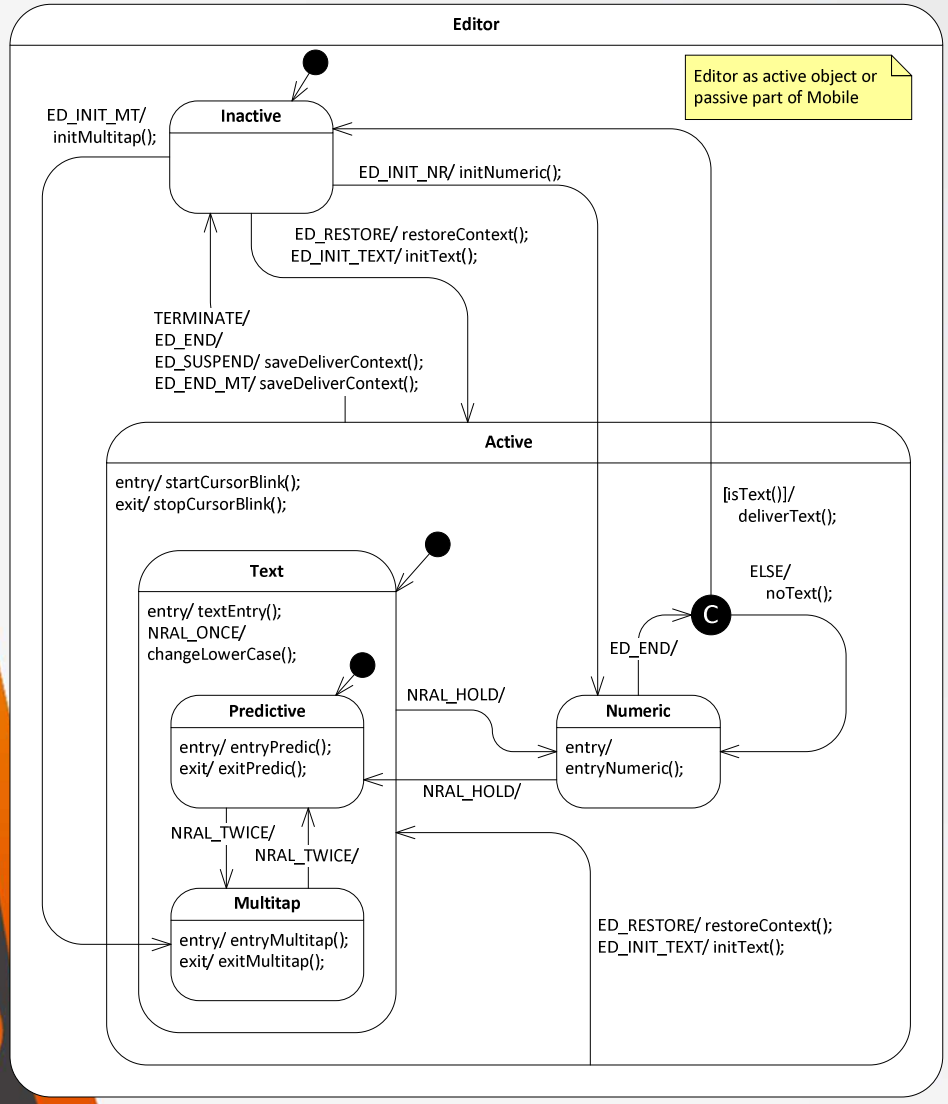


Editor como objeto activo
Mobile AO



- La primera aproximación es resolver la edición como un objeto activo.
- `Mobile` es el objeto activo que centraliza el control y gestión general del sistema.
- Este interactúa con el objeto activo `Editor`, el cual permite la edición de texto a partir de las teclas presionadas en el teclado del sistema.

Editor como objeto activo
Editor AO

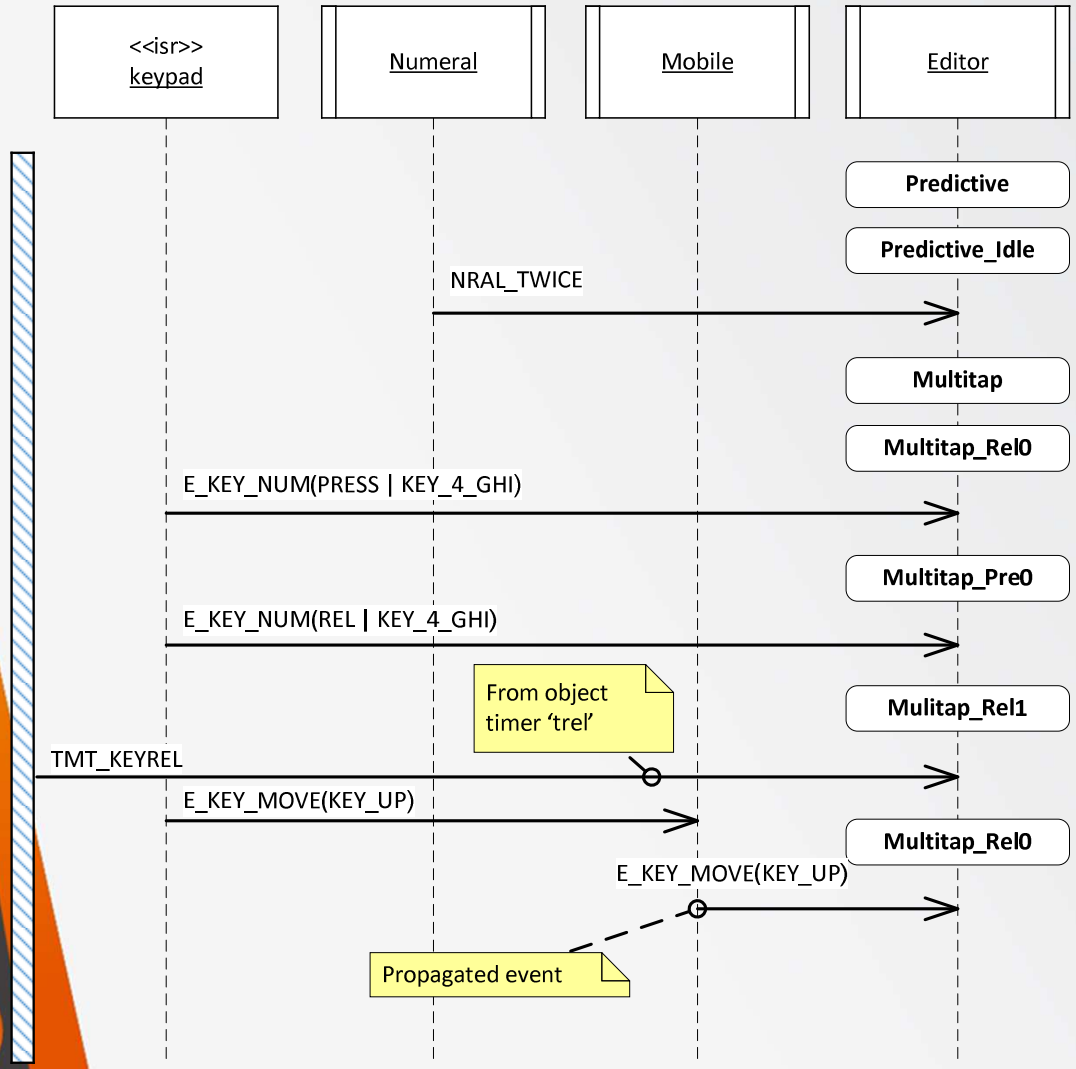


- El objeto activo Editor recibe eventos de `Mobile` y Numeral, y a su vez de la tarea dedicada al *scanning* del teclado, ejecutada en contexto de interrupción.
- Los objetos activos se comunican entre si mediante mensajes asincrónicos.

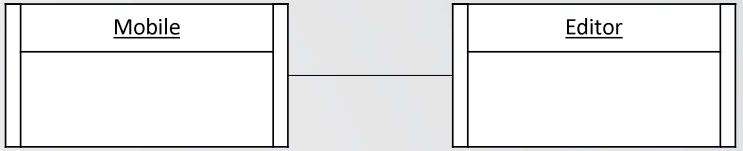
```
/* ---- Communication Mobile -> Editor ---- */
/* Editor as active object */
fwdToEditor(key)
{
  postFiFo(editor, key);
}
```

Editor como objeto activo

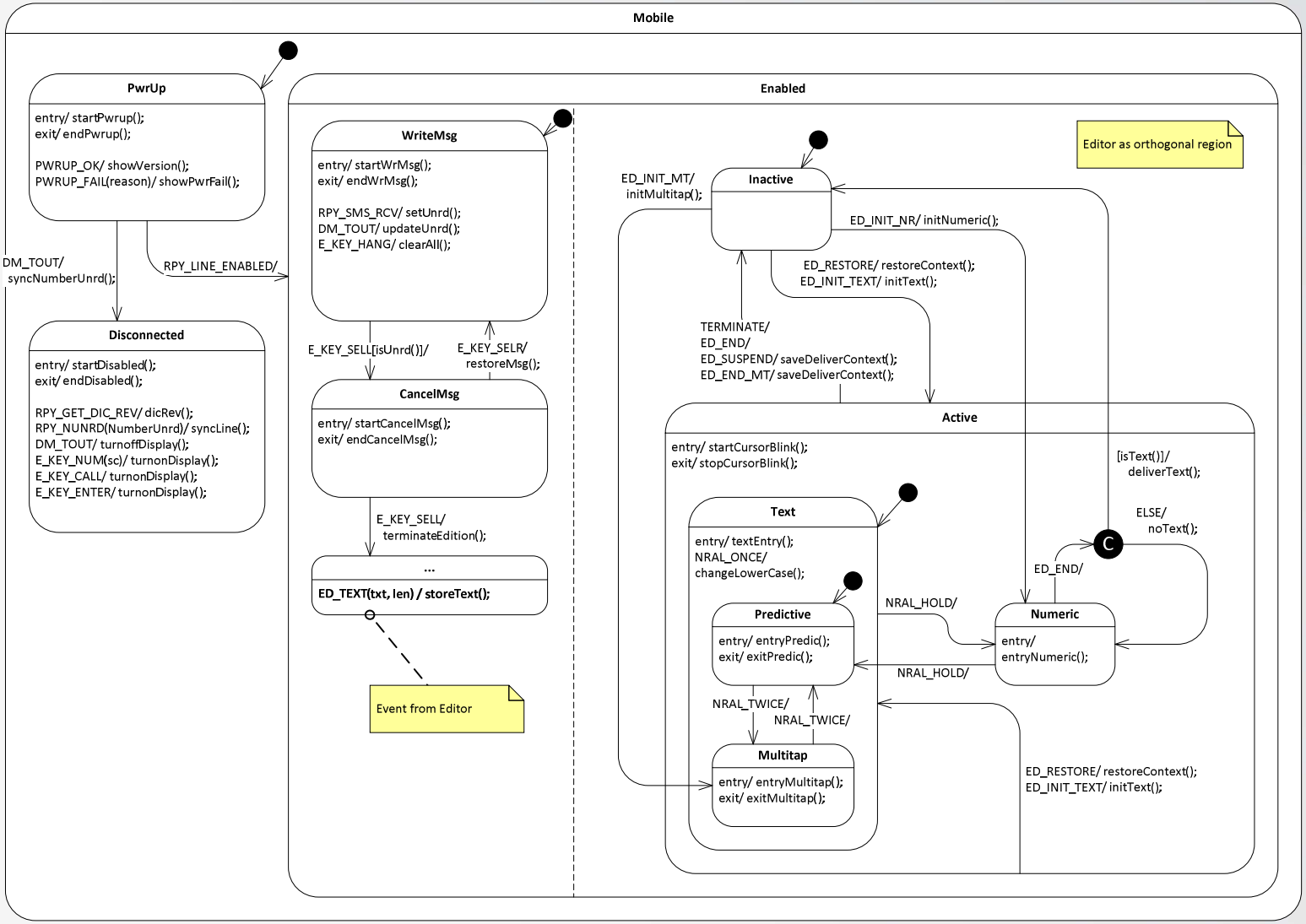
Interacción entre `Mobile` y `Editor`



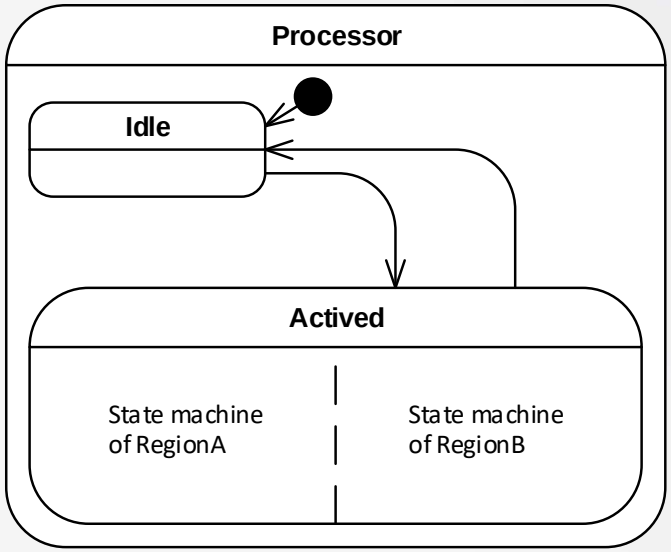
- El AO `Numeral` se encarga exclusivamente de resolver la funcionalidad de la tecla '#'.

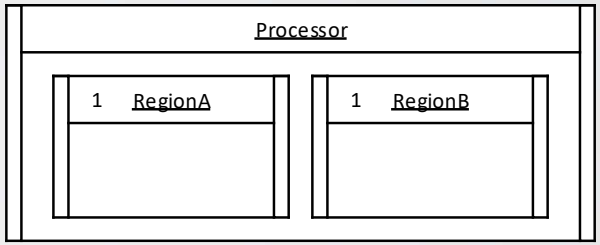


Editor como región ortogonal

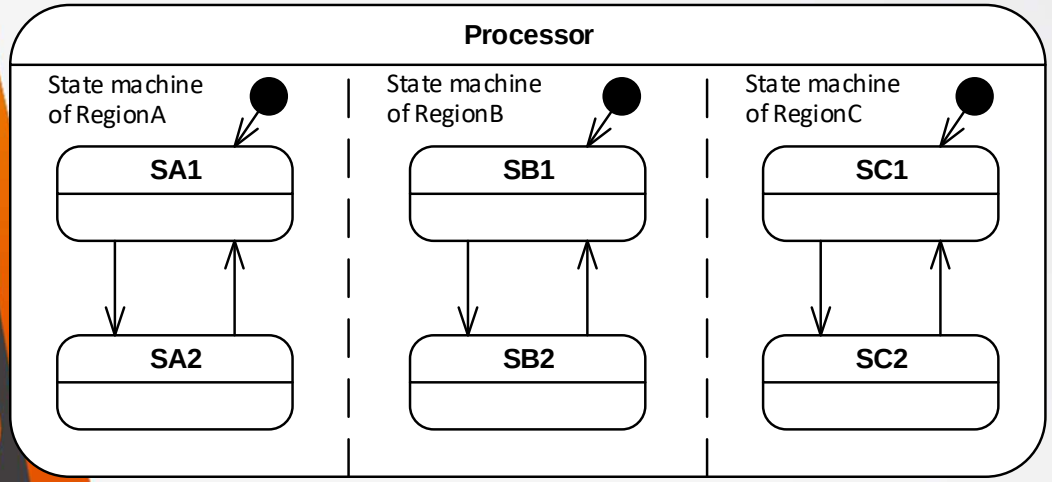


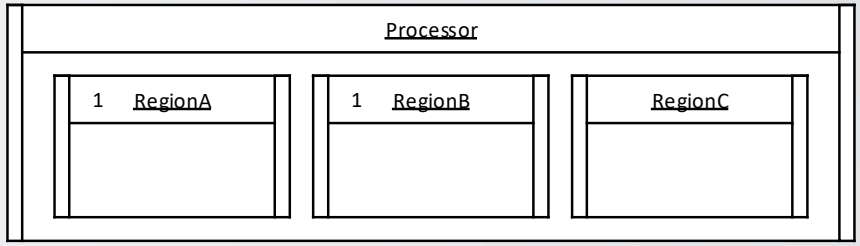
Objeto activo
Y las regiones ortogonales





- En ambos casos `Processor` posee una SM y esta se comunica con sus objetos pasivos que posee sus propias SM.





¿Estado ortogonal u objeto activo?

- Sabiendo que las regiones ortogonales:
  - Modelan procesadores de eventos independientes representados por máquinas de estados
  - Son concurrentes en el sentido lógico, y no necesariamente en el sentido de la ejecución
  - Estas pueden pertenecer a un estado compuesto o a una SM

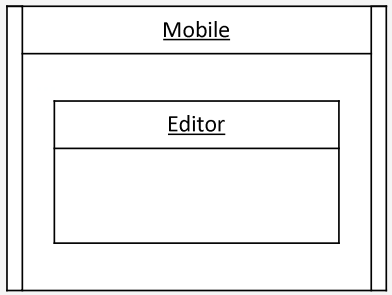
¿Qué limitaciones surgen si mi sistema modela cada proceso independiente con una región ortogonal?

- A estas no se les puede asignar individualmente una prioridad de ejecución, ya que heredan la prioridad de la SM que las contiene (o mejor dicho del AO), y por lo tanto no podríamos realizar un análisis de planificabilidad del sistema para determinar si el mismo alcanzará o no sus restricciones temporales.
- En ciertos sistemas esta falta de determinismo puede que sea invalidante.

Editor como objeto pasivo

```
/* ---- Communication Mobile -> Editor ---- */
/* Editor as passive part of Mobile */
fwdToEditor(me, key)
{
  dispatch(&me->itsEditor, key);
}
```

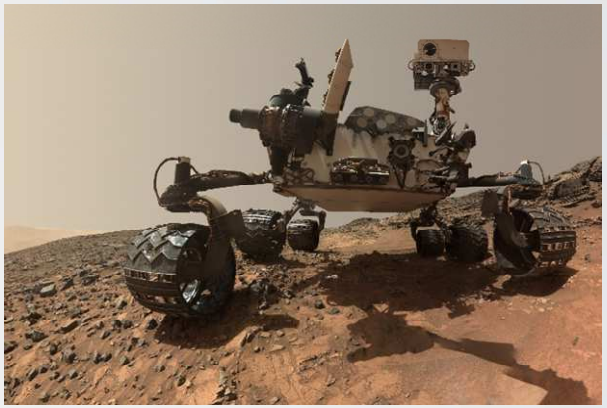
```
/* ---- Communication Editor -> Mobile ---- */
getEditedText()
{
  postLiFo(mobile, evText);
}
```



- `Editor` es un objeto pasivo con el mismo comportamiento que las soluciones anteriores.
- `Editor` se ejecuta en contexto del AO `Mobile`, de comportamiento similar a las soluciones anteriores.
- `Mobile` envía mensajes **sincrónicos** a `Editor`, con lo cual `Mobile` espera bloqueado la completitud del procesamiento de dicho mensaje.
- `Editor` le envía mensajes asincrónicos a `Mobile` mediante el mecanismo LIFO.

Objeto activo no reactivo
Periódico

- En ciertos casos el procesamiento de un objeto activo no está gobernado por eventos asincrónicos, sino más bien por un evento temporal periódico, o algún otro mecanismo de sincronización.
- En este caso el objeto activo permanece en reposo hasta que es señalizado para comenzar su ejecución.
- En algunas situaciones, el objeto activo se ejecuta periódicamente, y en cada ciclo de ejecución consume los eventos asincrónicos recibidos mientras se encontraba en reposo (*rate-driven* en lugar de *event-driven*), es decir, su comportamiento se vuelve *reactivo de a períodos*. Así está construido parte del software del Mars Rover Curiosity.
- Esto es útil para asegurar la planificabilidad de un sistema reactivo, logrando un sistema más determinístico.



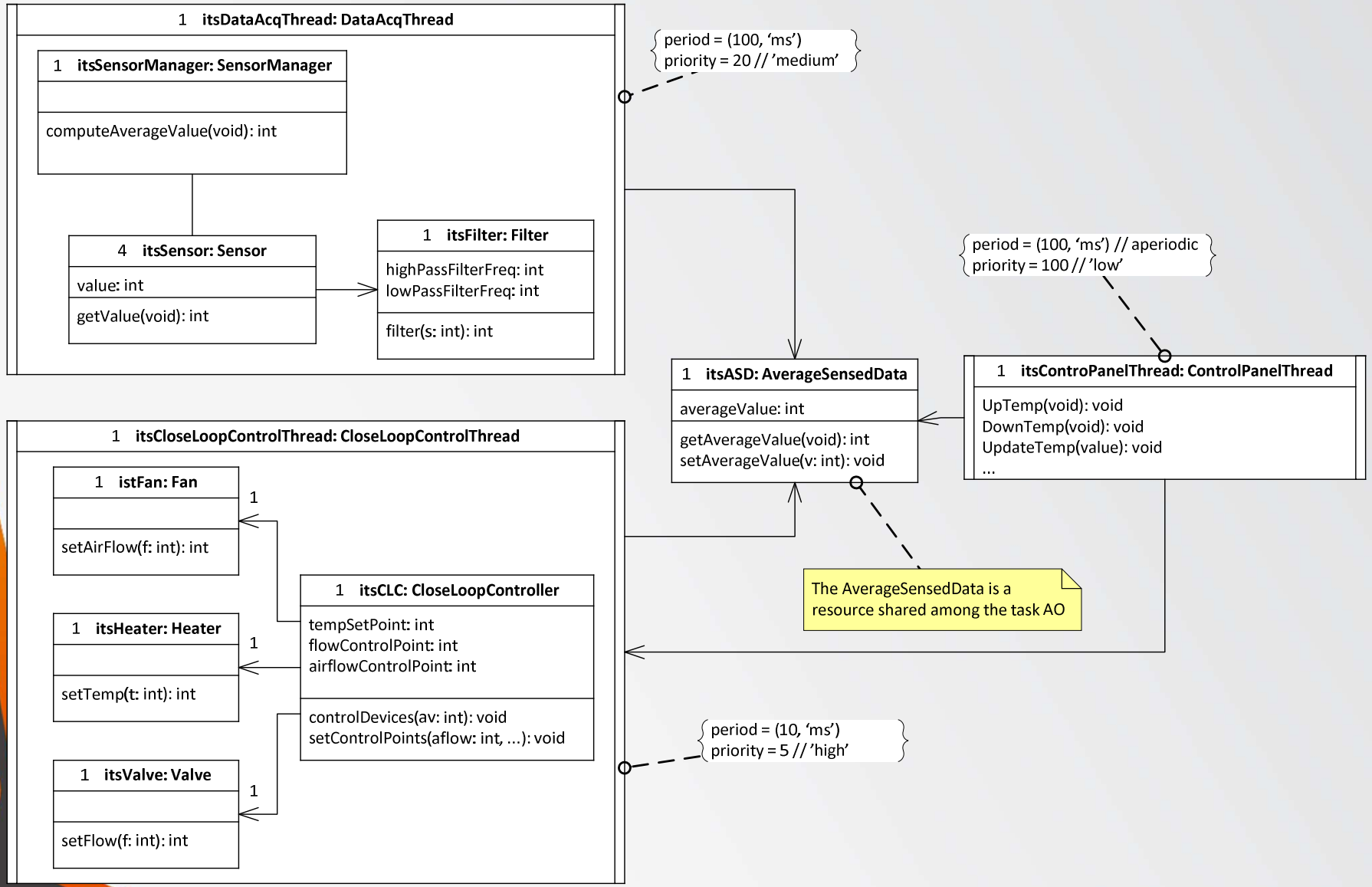
Objeto activo no reactivo
Continuo

- Un mismo sistema puede constituirse por diferentes partes concurrentes, las cuales pueden exhibir comportamientos reactivos y continuos, de esta forma, pueden coexistir y colaborar entre si, AO reactivos (con o sin SM) y AO continuos.
- En general, los AO de comportamiento continuo encapsulan un algoritmo, proceso o tarea particular independiente, posiblemente sobre un flujo de datos infinito, que requiere ejecutarse en su propio contexto de ejecución.
- Usualmente, su comportamiento actual (salida) depende tanto de sus salidas pasadas como de las entradas, pero esta dependencia es de naturaleza continua (o matemáticamente suave).
- Ejemplo son filtros digitales, lazos de control, sistemas de lógica difusa, etc.

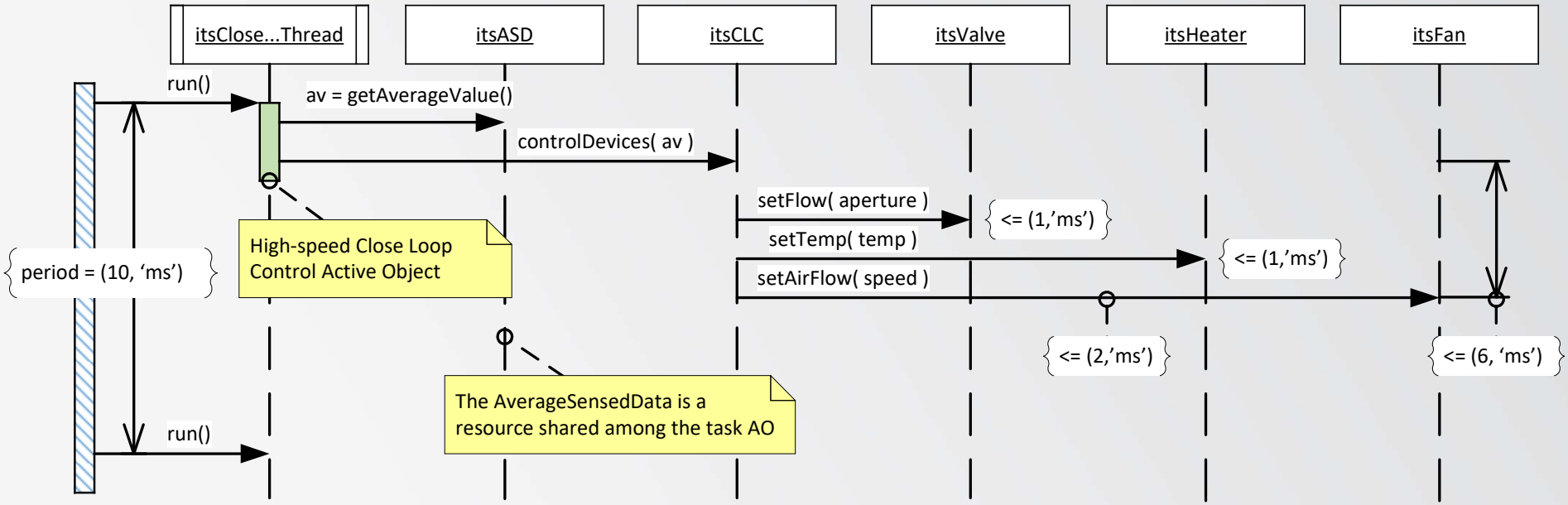
Colaboración entre AO continuos y reactivos

- Un dato es adquirido, filtrado y promediado desde cuatro sensores diferentes. La adquisición la realiza `SensorManager` a un ritmo de 10Hz (100ms). Este último utiliza el dato adquirido para actualizar y publicar `AverageSensedData.`
- Por su parte, `ControlPanel` supervisa todo el sistema recibe la actualización de la temperatura promedio censada y entre otras cuestiones puede ajustar los puntos de control del controlador de lazo cerrado de alta velocidad `CloseLoopControl`, el cual controla los actuadores `Fan`, `Heater` y `Valve` de acuerdo al dato promediado.
- `CloseLoopControl` actualiza la salida a 100Hz (10ms) y realiza el control de alta velocidad para reducir el error entre los valores promediados y los puntos de control.

Estructura

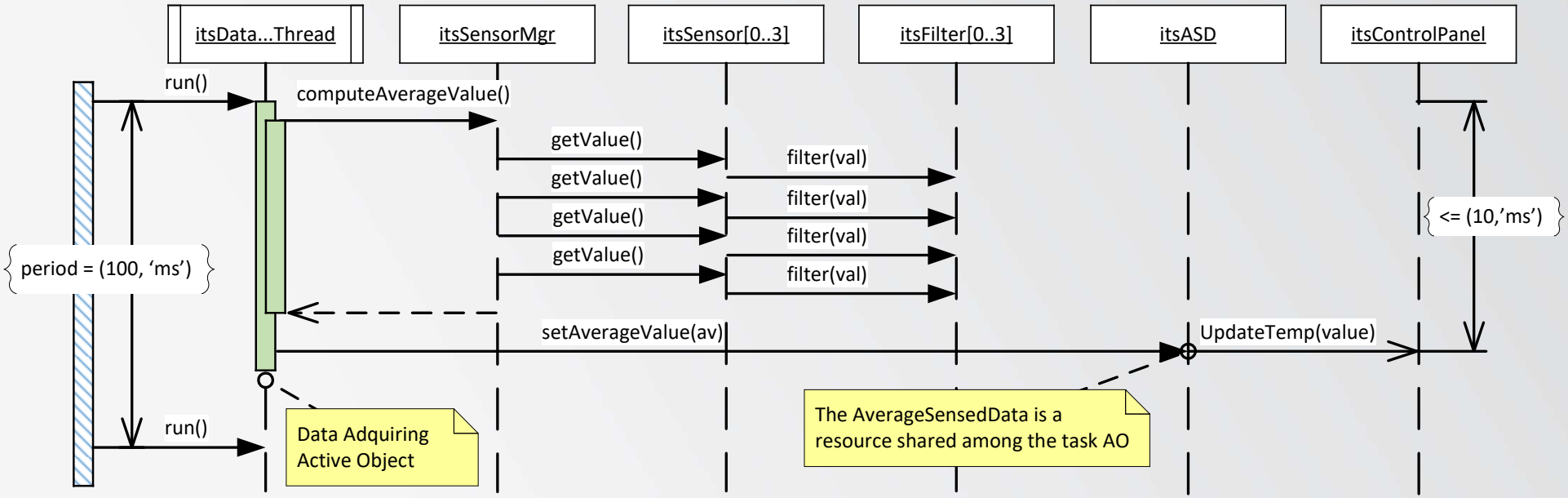

1 itsDataAcqThread: DataAcqThread
1 itsSensorManager: SensorManager
computeAverageValue(void): int
4 itsSensor: Sensor
value: int
getValue(void): int
1 itsFilter: Filter
highPassFilterFreq: int
lowPassFilterFreq: int
filter(s: int): int
period = (100, 'ms')
priority = 20 // 'medium'
period = (100, 'ms') // aperiodic
priority = 100 // 'low'
1 itsASD: AverageSensedData
averageValue: int
getAverageValue(void): int
setAverageValue(v: int): void
1 itsControPanelThread: ControlPanelThread
UpTemp(void): void
DownTemp(void): void
UpdateTemp(value): void
...
The AverageSensedData is a resource shared among the task AO
1 itsCloseLoopControlThread: CloseLoopControlThread
1 istFan: Fan
setAirFlow(f: int): int
1 itsHeater: Heater
setTemp(t: int): int
1 itsValve: Valve
setFlow(f: int): int
1 itsCLC: CloseLoopController
tempSetPoint: int
flowControlPoint: int
airflowControlPoint: int
controlDevices(av: int): void
setControlPoints(aflow: int, ...): void
1
1
1
period = (10, 'ms')
priority = 5 // 'high'

Interacción



- Ciclo de ejecución del AO `CloseLoopControl`.

Interacción



- Ciclo de ejecución del AO `DataAcq`.

Implementación
DataAcq y CloseLoopController

```
/* DataAcq.c */

static int
computeAverageValue(DataAcq *const me)
{
  …
  for (av = 0; sensor = me->itsSensor; …)
     av += Sensor_getValue(sensor);
  return av;
}

static ...
taskFunction(...)
{
  …
  while (1)
  {
    OS_sleep(100);
    av = computeAverageValue();
    AverageSensedData_setAverageValue(av);
  }
}
```

```
/* CloseLoopController.c */

static void
controlDevices(int av)
{
  …
  Valve_setFlow(aperture);
  Heater_setTemp(temp);
  Fan_setAirFlow(speed);
}

static ...
taskFunction(...)
{
  …
  while (1)
  {
    OS_sleep(10);
    av = AverageSensedData_getAverageValue();
    controlDevices(av);
  }
}
```

- ¿Cómo podría convertir estos AO en reactivos utilizando SMs?

Implementación
AverageSensedData y ControlPanel

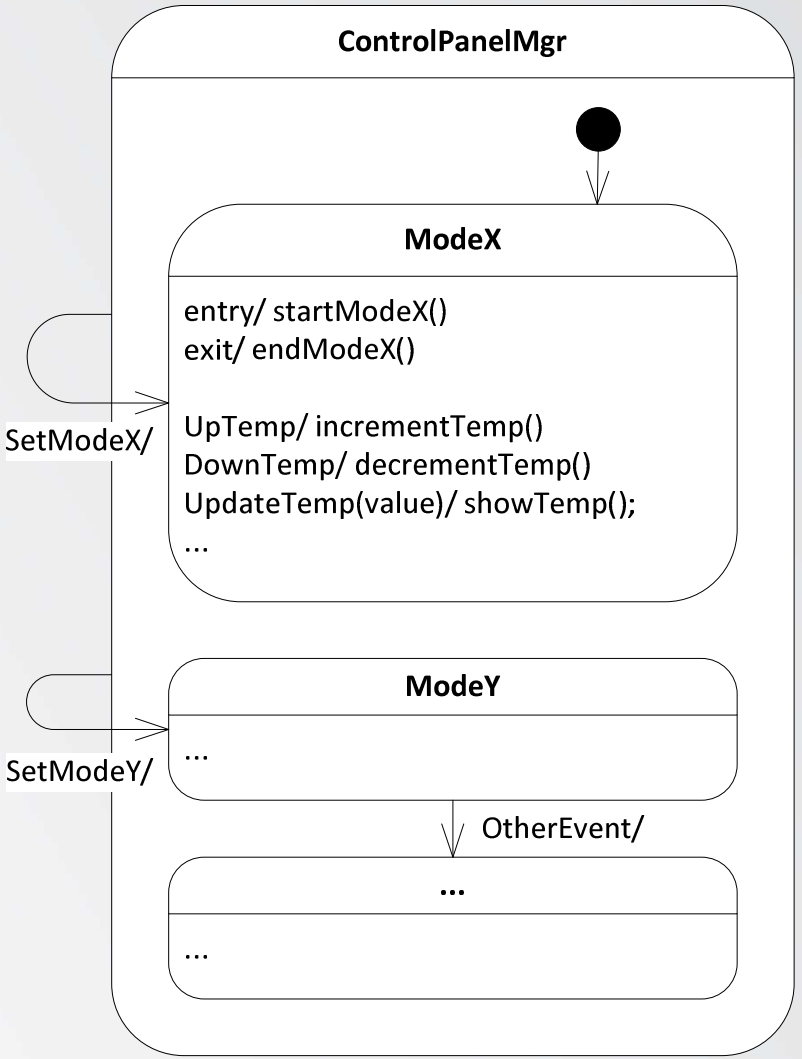
```
/* AverageSensedData.c */

static int averageSensedData;

void
AverageSensedData_setAverageValue(int av)
{
  OS_enterCritical();
  averageSensedData = av;
  OS_exitCritical();
  …
  evUpdateTemp.value = averageSensedData;
  OS_QueuePost(ControlPanel, &evUpdateTemp);
}
```

```
/* ControlPanel.c */

static ...
incrementTemp(...)
{
  ...
  CloseLoopController_setControlPoints(…);
}
```



Implementación alternativa
Eliminando el recurso compartido

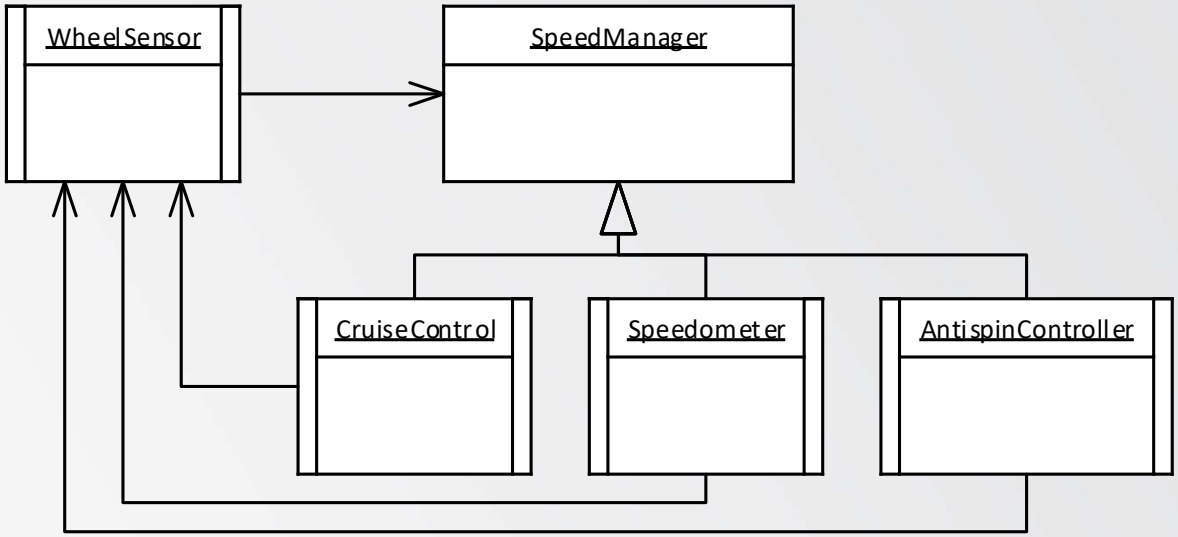
```
/* CloseLoopController.c */

static ...
taskFunction(...)
{
  …
  while (1)
  {
    OS_sleep(10);
    ++timeCounter;

    if (timeCounter == 10)
    {
      av = computeAverageValue();
      AverageSensedData_setAverageValue(av);
    }
    controlDevices(av);
  }
}
```

- Ya que los períodos de ejecución de `CloseLoopController` y `DadaAcq` son múltiplos enteros (10/1), ambos pueden ejecutarse en un mismo contexto, con el período más rápido de ambos, evitando así compartir el valor censado promedio en un ambiente concurrente.

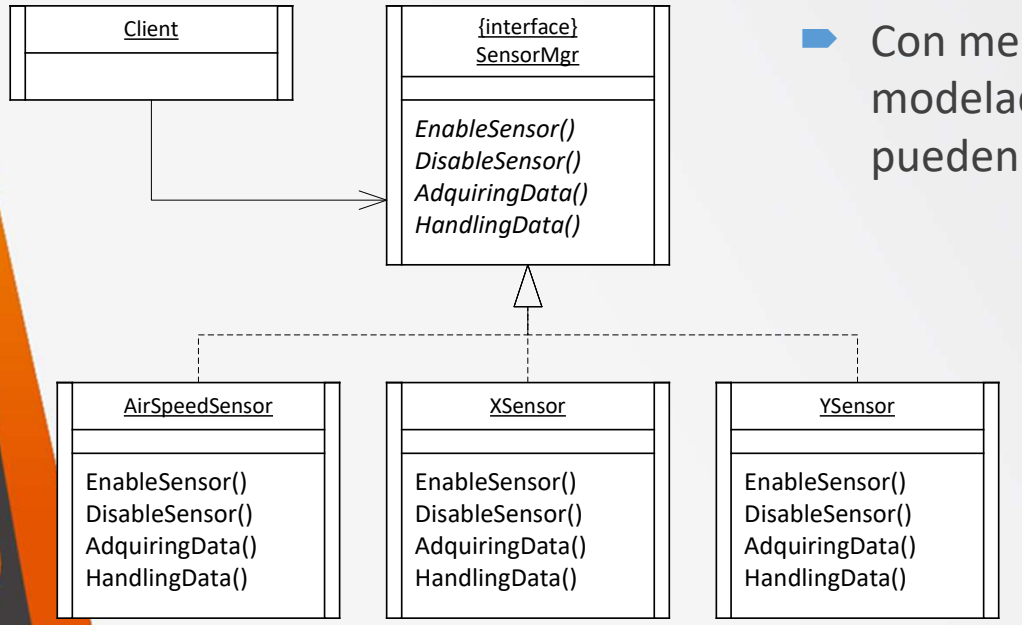
Objeto Activo
Patrón observador



- `CruiseControl`, `Speedometer`, `AntispinController` (observadores) se subscriben para recibir las actualizaciones de `WheelSensor` (servidor).
- Los observadores reciben la actualización desde el servidor de manera asincrónica (push) evitando el mecanismo de encuesta (polling).
- La subscripción/de-subscripción se efectúa en tiempo de ejecución
- El servidor está **desacoplado** del sus observadores.

Objeto activo polimórfico

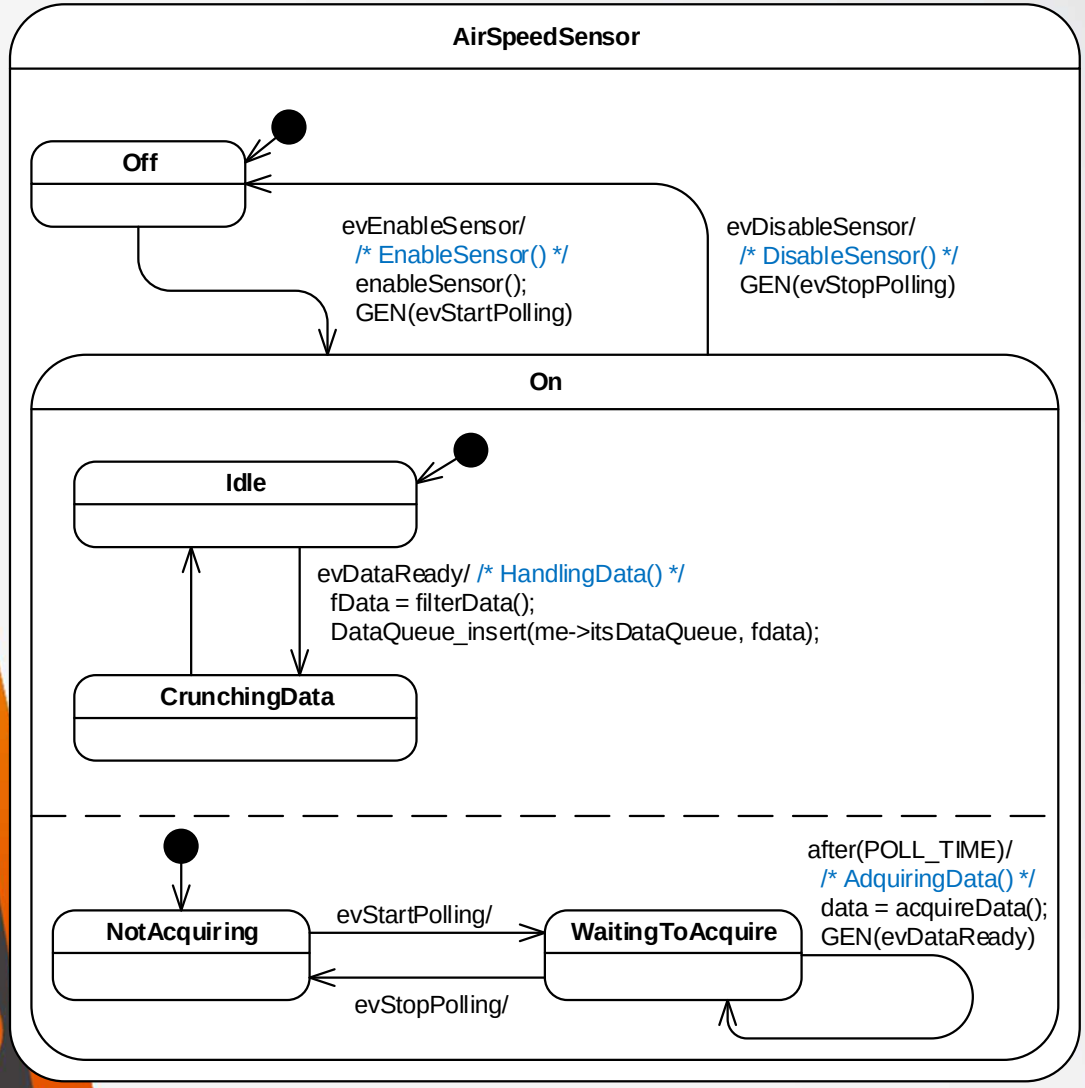
- El comportamiento de un AO puede ser polimórfico, en cuyo caso puede adoptar diferentes funcionalidades (formas) de acuerdo al contexto de operación. Esto es una abstracción.
- El polimorfismo puede aplicarse tanto en la recepción de mensajes sincrónicos (llamadas a métodos) como en la recepción de mensajes asincrónicos.



- Con mensajes asincrónicos y comportamiento modelado por SM, las acciones de esta última pueden ser polimórficas.

```
/* SensorMgr.c */
/* Effect action */
SensorMgr_EnableSensor(SensorMgr *const me)
{
  (*me->vptr->EnableSensor)(me);
}
```

Objeto activo polimórfico




```
/* SensorMgr.c */
/* Effect action */
SensorMgr_EnableSensor(SensorMgr *const me)
{
  (*me->vptr->EnableSensor)(me);
}
```

- `EnableSensor()`, `DisableSensor()`, `HandlingData()` y `AdquiringData()` son acciones de efecto polimórficas

Framework de tiempo-real

¿Porqué un framework?

- Sabiendo que:
  - La mayor parte de la infraestructura entre aplicaciones del mismo tipo es la misma.
  - Usualmente, entre el 60% y 90% de una aplicación es código ya implementado, que podría reutilizarse estructurándolo adecuadamente.
- Si se capturan las funcionalidades comunes requeridas por las aplicaciones de un mismo tipo, de forma tal que, puedan *reutilizarse* como están, *especializadas* según corresponda o fácilmente *reemplazadas* si es necesario, se construye un conjunto de funcionalidades que constituye el esqueleto de la aplicación.

ESTA INFRAESTRUCTURA SE LA DENOMINA FRAMEWORK

Beneficios de un framework

- Minimiza la complejidad de las aplicaciones
- Reduce drásticamente el tiempo de lanzamiento al mercado de nuevos productos y los costos de mantenimiento
- Focaliza los equipos de desarrollo en conseguir los requerimientos funcionales específicos del producto
- Sus actualizaciones mejoran la aplicación sin esfuerzo adicional
- Promueve la adopción de técnicas comunes entre los desarrolladores
- Promueve el desarrollo de código limpio de alta calidad

Herramientas y frameworks

StateFlow

BridgePoint

IBM Rational Rhapsody

Historia

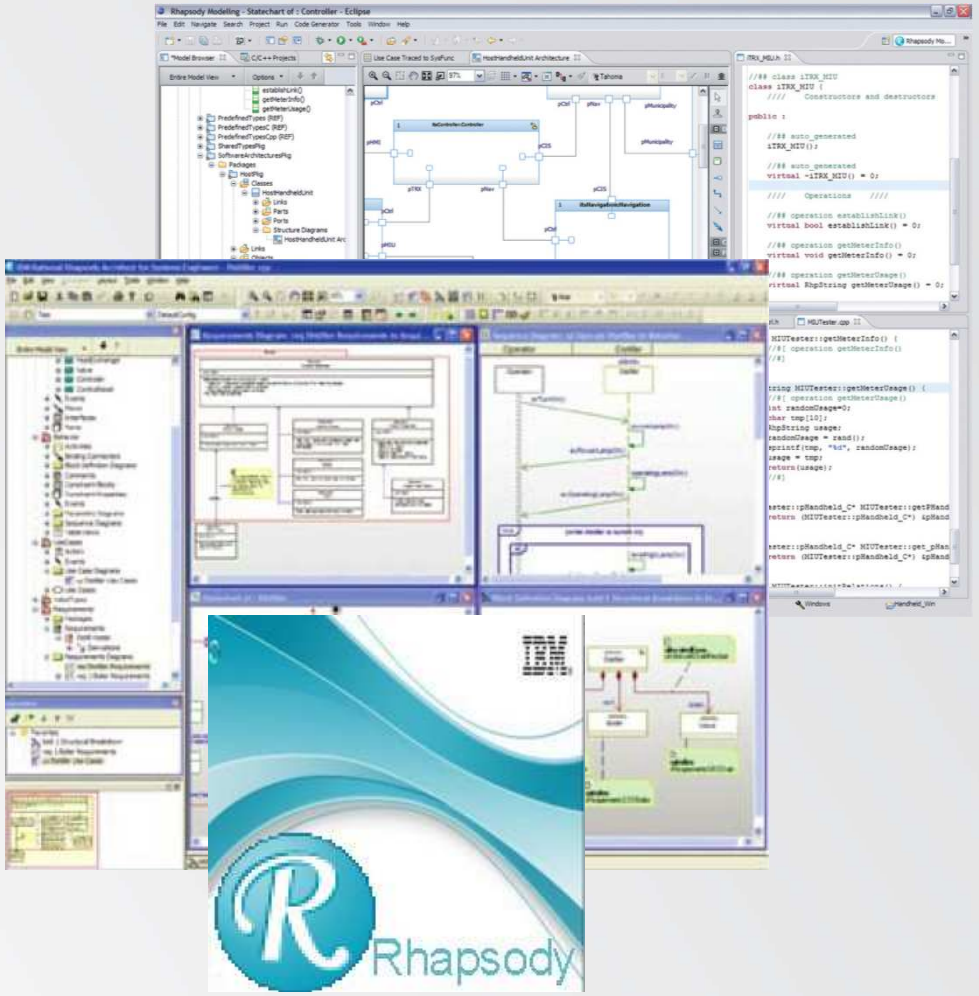
1984 - Davir Harel y cia, especifican el formalismo *statechart*, durante una consultoría del proyecto Lavi Aircraft

1986 – Estos desarrollan Statemate, para modelar y generar código, primero en Ada y luego en C, para RTES, fundando I-Logix

1996 – Lanzan Rhapsody, una herramienta como Statemate pero orientada a objetos.

2006 – Telelogic AB adquiere I-logix

2008 – IBM adquiere Telelogic AB y Rhapsody es parte de sus productos

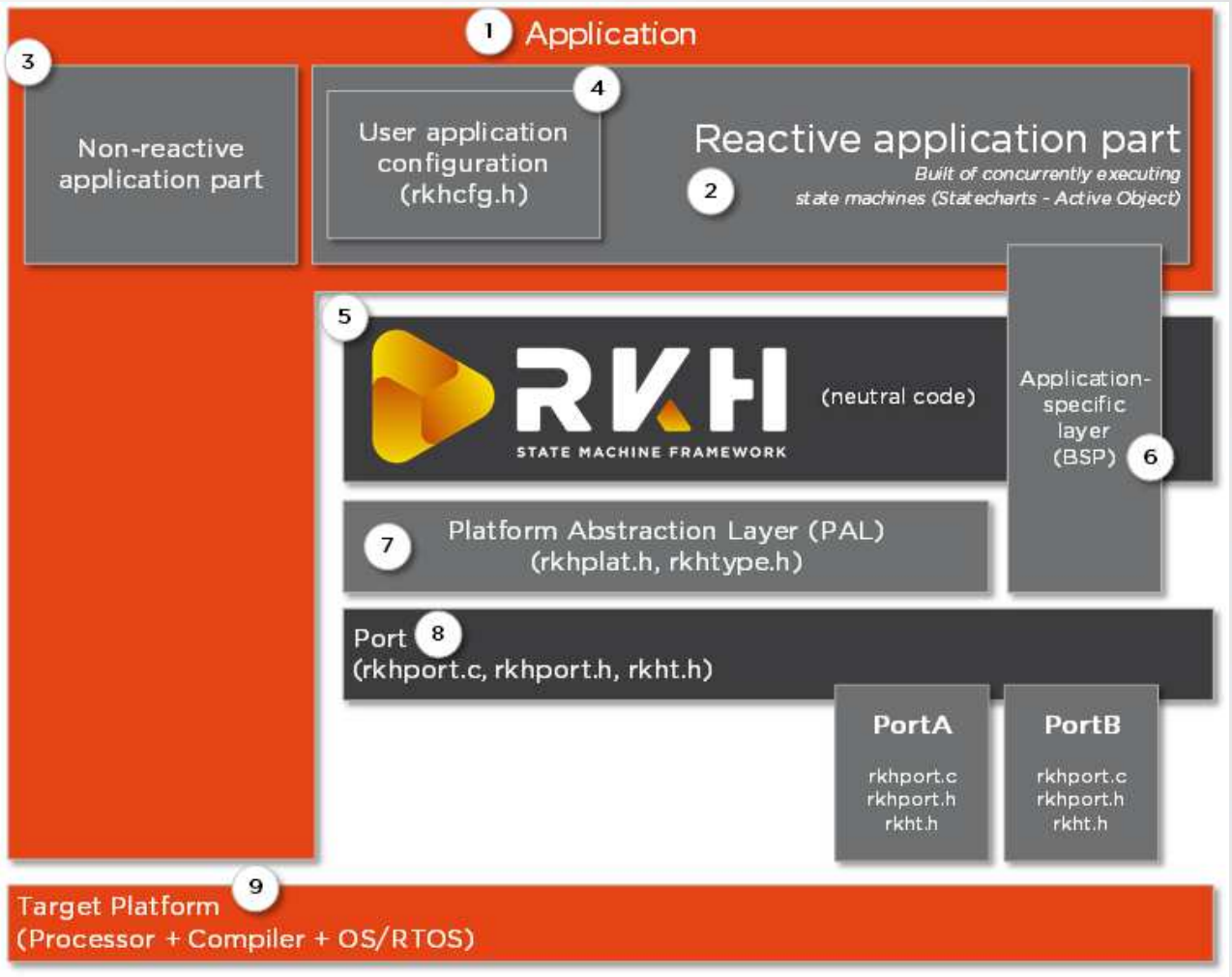


RKH

AGENCIA NACIONAL DE PROMOCION CIENTIFICA Y TECNOLOGICA

www.vortexmakes.com

Estructura


1 Application
3
Non-reactive application part
4
User application configuration (rkhcfg.h)
Reactive application part
Built of concurrently executing state machines (Statecharts - Active Object)
2
5
RKH
STATE MACHINE FRAMEWORK
(neutral code)
Application-specific layer (BSP)
6
7
Platform Abstraction Layer (PAL) (rkhplat.h, rkhtype.h)
Port
8
(rkhport.c, rkhport.h, rkht.h)
PortA
rkhport.c
rkhport.h
rkht.h
PortB
rkhport.c
rkhport.h
rkht.h
9
Target Platform
(Processor + Compiler + OS/RTOS)

Preguntas

Referencias

[1] RKH, “RKH Sourceforge download site,”, http://sourceforge.net/projects/rkh-reactivesys/ , August 7, 2010.

[2] Object Management Group, “Unified Modeling Language: Superstructure version 2.1.1,” formal/2007-02-05, Febrary 2007.

[3] Herb Sutter, “Prefer Using Active Objects Instead of Naked Threads”, Dr.Dobb’s, June 14, 2010.

[4] Herb Sutter, “Use Threads Correctly = Isolation + Asynchronous Messages”, Dr.Dobb’s, March 16, 2009.

[5] David Cummings, “Managing Cuncurrency in Complex Embedded Systems”, Fujitsu Laboratories Workshop on Cyber-Physical Systems, June 10, 2010.

[6] R. Greg Lavender, Douglas C. Schmidt, “An Object Behavioral Pattern for Concurrent Programming”, 1996.

[7] B. P. Douglass, “Real-Time UML: Advances in the UML for Real-Time Systems (3rd Edition),”, Elseiver, October 4, 2006.

[8] M. Samek, “Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems,” Elsevier, October 1, 2008.

[9] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Sci. Comput. Programming 8 (1987), pp. 231–274.

[10] L. Francucci, “Modelos y frameworks en embedded software de manera simple”, June 01, 2017