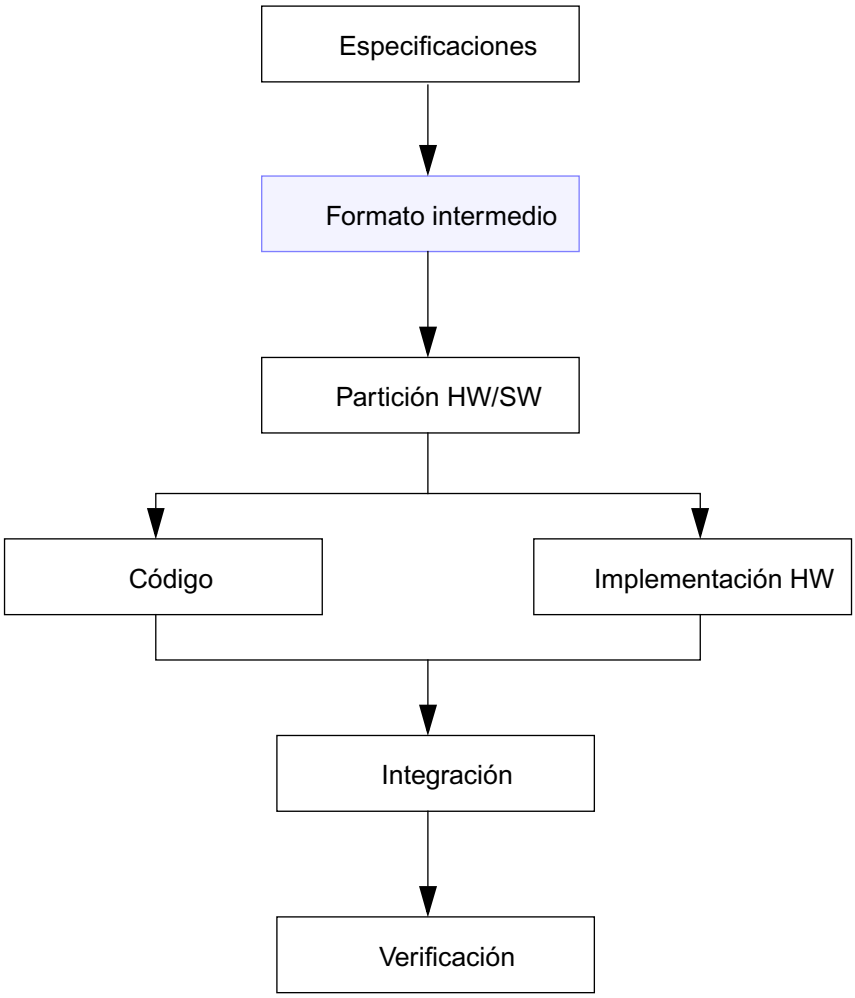
Tema II

Modelos de Especificación y Lenguajes

En este tema se va a tratar de dar una especificación al comportamiento de los sistemas, de tal forma que podamos describirlo así como tener un entorno para poder obtener el comportamiento deseado.



2.1. Introducción. Modelado del problema

Estamos en la primera etapa del flujo de diseño, en la cual tenemos que dar un modelo a los procesos y algoritmos que implementan el comportamiento que queremos implementar. Por lo tanto, el comportamiento del futuro diseño se debe abstraer desde una descripción física (un esquema de componentes electrónicos que conforman el diseño del sistema), es decir, se debe describir el comportamiento desde el nivel de sistema. Esta descripción se ajusta a un modelo de computación, ya que es necesario describir cómo las señales de salida son generadas a partir de las señales de entrada.

Un modelo de computación se puede definir de la siguiente forma:

> Un modelo de computación es un entorno de trabajo en el que expresar la secuencia de acciones que deben ser ejecutadas para completar una operación.

Como ejemplos de modelos de computación podemos indicar las máquinas de estado finito, redes de Petri, etc. En contraposición, tenemos las funciones que son las relaciones existentes entre las señales de entrada y de salida. Vamos a distinguir entre funciones y modelos de computación a través del ejemplo de un multiplicador (aunque sea un sistema excesivamente simple para ser implementado en un sistema empotrado). La función de un multiplicador será realizar la multiplicación de dos datos de entrada; en cambio, el modelo de computación asociado indicará el momento y la forma de realizar cada una de las operaciones asociadas a esa función.

A la hora de tratar el modelado de un comportamiento, es necesario distinguir entre modelado y lenguaje. Mientras que el modelo de computación necesita ser lo bastante poderoso para el dominio de aplicación y gozar de los algoritmos de síntesis y validación apropiados, el lenguaje necesita ser lo bastante expresivo en el dominio de aplicación y tener sentido en el modelo de computación deseado. Luego, el lenguaje será utilizado para la descripción del sistema, y el modelo de computación será el que interprete y le dé sentido a la descripción.

Una táctica ampliamente utilizada consiste en independizar las señales de dato y de control, de tal forma que ambos tipos de señales tendrán flujos independientes. El motivo de esta separación radica en que

- las señales de control
  - no saben cuando llegan las señales de dato,
  - y el tiempo de llegada de las señales de control pueden tener valores diferentes
- las señales de dato
  - suelen llegar en bloques regulares (conocidos como muestras)

Esta separación puede ser utilizada en los diferentes métodos de especificación, síntesis y validación de sistemas. Estos métodos prestarán más atención a determinadas características, en función del tipo de señal que se esté tratando. Así en la tabla 2.1 se muestra el énfasis de cada uno de los métodos según el tipo de señal. Por ejemplo, en los métodos de validación, todos los procesos sobre señales de datos tendrán el mismo tratamiento y prioridad, mientras que sobre las señales de control no será así puesto que ellas impondrán que proceso debe realizarse antes.

	Señales de dato	Señales de control
Especificación	Dependencia funcional entre entradas y salidas	Relación causa/efecto
Síntesis	Eficiencia memoria/tiempo (pipeline)	Tiempo de respuesta
Validación	Todos los procesos y eventos son iguales	Prioridad entre eventos y procesos.

Tabla 2.1. Énfasis de los diferentes métodos según el tipo de señal considerada.

Seguidamente, y antes de empezar con los diferentes tipos de modelos, vamos a desglosar las principales propiedades que tendrá un sistema, definiendo

> una propiedad como una afirmación (y no como una descripción) del comportamiento de un diseño

las cuales suelen ser redundantes (con las especificaciones del sistema). El hecho de la separación con respecto a las especificaciones del diseño radica en que las propiedades son más simples, abstractas e incompletas. Por ejemplo, una especificación podría ser que el sistema no llegue a un estado *muerto* desde el cual no se pueda salir; en cambio la propiedad sería *liveness* (no tener ningún estado muerto).

2.1.1. Concurrencia

La propiedad de concurrencia se puede definir de la siguiente forma

> La concurrencia es la propiedad de ejecutar varias tareas paralelamente en el tiempo.

Esta propiedad es importante ya que los sistemas suelen estar formados por un conjunto de comportamientos concurrentes. Ésta puede darse a diferentes niveles de abstracción, como pueden ser niveles de trabajo, tareas, operación, bit, etc.

Veamos el diagrama de la figura 2.1. En ella podemos ver cuatro tareas diferentes (dos sumas, una multiplicación y una resta). Todas estas operaciones serán realizadas cuando lleguen los respectivos datos de entrada, generando así un determinado orden de ejecución. Las primeras tareas que se realizarán serán una suma y una resta (las cuales pueden ser realizadas de forma concurrente), después de estas operaciones será realizada la mutliplicación, y finalmente se realizará la segunda suma. Así la concurrencia de tareas nos dará idea de los hilos de control que tendrá el sistema,

> entendiendo por hilo de control a un conjunto de operaciones que se ejecutarán de forma secuencial

Así en la figura tendremos seis hilos formados por sumador-Q, sumador-multiplicador-P, restador-multiplicador-P, sumador-multiplicador-sumador-R, restador-multiplicador-sumador-R y restador-sumador-R. Aquí hemos separado los hilos en función de las señales de salida que podamos considerar: Q, P y R

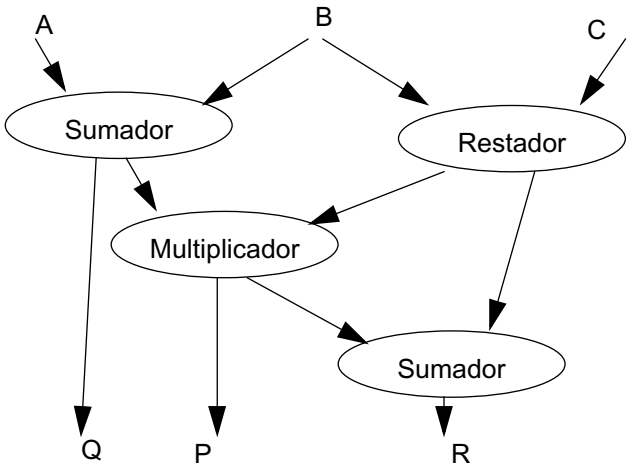


Figura 2.1.- Ejemplo de diagrama de comportamiento.

2.1.2. Transiciones de estado

La mayoría de los sistemas (sino todos) pueden verse divididos en estados, los cuales pueden ser un modo o etapa, o bien una computación. Así, en la figura 2.1 podríamos dividir el comportamiento en cuatro estados diferentes (uno por cada operación que se debe realizar).

Luego, el comportamiento no será más que el flujo de los datos a través de posibles estados del sistema. Este flujo no tiene porqué recorrer todos los estados, ya que dependerá en gran medida del valor de los datos.

2.1.3. Jerarquía

El trabajo jerárquico es necesario en el diseño o análisis de un sistema ya que pasaremos de un problema complejo a varios problemas menos complejos. De esta forma se puede centrar la atención en las tareas más concretas sin necesidad de considerar el sistema completo.

Siguiendo con el ejemplo de la figura 2.1, se ha dividido el problema en tres comportamientos: suma, multiplicación y resta. Esta jerarquización no tiene porqué ser de elementos hardware, sino que también se suele jerarquizar los elementos software, es decir, el código.

2.1.4. Construcciones de programación

Esta propiedad es básica en los sistemas empotrados, en los que parte del comportamiento se implementa vía hardware y parte vía software. Este hecho viene reforzado con que algunos comportamientos son más fácilmente implementables con algoritmos secuenciales vía software que vía hardware.

Dentro de estas construcciones podemos encontrar diversas estructuras como son asignamientos, bucles, subprogramas, tipos de datos complejos (como registros o listas).

2.1.5. Acabado

Esta propiedad implica que todas las operaciones deben poder ser realizadas. Esta propiedad permite la jerarquización del comportamiento ya que al suponer que la operación anterior va a ser realizada de forma completa, se puede centrar la atención en la nueva tarea (operación) suponiendo que los datos van a llegar completamente válidos.

Según la figura 2.1, para que el resultado de la multiplicación sea correcta, tanto la operación del primer sumador como del restador deben poder realizarse de forma completa, y una vez hayan acabado éstas, se podrá empezar la multiplicación.

2.1.6. Comunicación

La comunicación es una propiedad transcendental en el diseño de sistemas empotrados ya que los diferentes bloques (tanto hardware como software) del sistema deben intercambiar datos entre sí. La comunicación debe ser correcta debido a la propiedad de acabado puesto que si la comunicación de datos no es correcta, la propiedad de acabado estará falseada, porque aunque las operaciones hayan finalizado, los datos comunicados no serán los últimos o no estarán acabados.

Con respecto a esta propiedad podemos encontrarnos con varias estrategias a seguir. Entre las estrategias más comunes podemos destacar las siguientes:

- Comunicación desincronizada. El productor y consumidor de información no están coordinados. Por lo tanto, no existe ninguna garantía de que la comunicación entre ambos se dé de forma correcta, es decir, puede ocurrir que el consumidor coja unos datos anteriores o que el productor machaque los datos antes que el consumidor los capture.
- Comunicación “lectura-modificación-escritura”. En esta estrategia existe una llave que evita la sobrescritura o la lectura previa ya que se debe completar el ciclo anterior. Esta estrategia es muy utilizada en el acceso a datos distribuidos.
- Comunicación a través de FIFO ilimitada. En esta estrategia, el consumidor tiene a su entrada una memoria FIFO en la que el productor de información va almacenando la información y el consumidor la va capturando a medida que es necesaria (ya habiendo sido generada). Esta estrategia es un método de comunicación punto a punto.
- Comunicación a través de FIFO limitada. Esta estrategia es la misma que la anterior, excepto que el tamaño de las FIFOs de entrada tienen un valor limitado. A veces se necesita otro elemento para evitar el desbordamiento de las FIFOs.
- Comunicación *rendezvous* (por asignación). En esta estrategia, el productor produce la información y el consumidor la captura en el mismo punto del flujo de control (lo que se entiende por asignación de señal).

En el ejemplo de la figura 2.1 se podría utilizar cualquier tipo de comunicación, pero lo realmente importante es que las operaciones productoras (por ejemplo, la primera suma y resta) evalúen sus datos de entrada (den la información) y la operación consumidora (la multiplicación) pueda capturarla sin perder ningún dato, y todos los datos capturados sean válidos.

2.1.7. Sincronización

Esta propiedad está íntimamente relacionada con la anterior, y consiste en el momento en que los diferentes bloques deben comunicarse. El principal problema se encuentra en la heterogeneidad de los elementos que componen el sistema completo, y por lo tanto, la diferencia de las velocidades de operación de cada uno de ellos.

La sincronización no es requerida en todo momento de operación, sino únicamente en momentos determinados. Estos momentos vienen determinados por el intercambio de datos entre diferentes partes (ya que las diferentes partes deben operar con los datos correctos), y las diferentes actividades que deben ser realizadas simultáneamente.

2.1.8. Temporización

Esta propiedad es requerida para representar las implementaciones del mundo real (sobre todo para los sistemas interactivos y reactivos).

De hecho, cada tipo de modelado considera al tiempo desde perspectivas diferentes. Podemos considerar al tiempo como una medida continua o como una medida discreta.

2.2. Elementos del modelo de computación

En esta sección se presentará un entorno para comparar los elementos de diferentes modelos de computación, llamado modelo *tagged-signal*, y será utilizado para contrastar diferentes estilos de comportamiento secuencial, concurrencia y comunicación.

2.2.1. Modelo *tagged-signal*

Este modelo ha sido propuesto por Lee y Sangiovanni-Vicentelli como un formalismo para describir aspectos de modelos de computación para especificaciones de sistemas empotrados.

En primer lugar debemos definir los componentes fundamentales de este modelo, los cuales son *tag*, eventos, señales y procesos.

- Un *tag* es una marca temporal de un sistema físico. No obstante, en los sistemas con un elevado grado de concurrencia, no es fácil mantener el orden temporal determinado por los *tags*, así que su definición se vuelve más abstracta dando ordenamientos parciales.
- Un evento se define como un par valor/tag, dentro del conjunto de valores y de tags.
- Una señal es un conjunto de eventos. Para que se pueda decir que la señal sea funcional o apropiada, se debe cumplir una relación biunívoca entre el tag y el valor de todos los eventos, es decir,

$$\forall evento \rightarrow e_1 = (t, v_1), e_2 = (t, v_2) \Rightarrow v_1 = v_2$$

También suele ser útil definir un vector de señales (un bus) como un conjunto de ellas.

- Un proceso es un conjunto de posibles comportamientos o una relación entre señales (de entrada y salida). Un proceso se denomina determinado cuando existe un único comportamiento para un determinado conjunto de valores de las señales de entrada; y en caso contrario se denominará indeterminado. Todos los procesos funcionales serán determinados.

Dentro de cualquier modelo de un sistema físico, una de las variables más importantes es el tiempo. En el modelo que nos ocupa, el tiempo viene determinado por los *tags* de los eventos. En función del tratamiento que se le dé al tiempo podemos encontrar sistemas temporizados y no temporizados.

Una posible definición de sistema temporizado es la siguiente

Un sistema se define como temporizado cuando el conjunto de todos los tags del sistema están ordenados,

por lo tanto existe una línea temporal fija. Para poder realizar operaciones con los *tags*, se debe definir una métrica con el conjunto de los *tags*. Este conjunto es un grupo abeliano, por lo que se puede definir una operación cerrada denominada suma y existe el elemento neutro y opuesto.

En este tipo de sistemas podemos encontrarnos dos tratamientos diferentes con respecto al tiempo. Por un lado tenemos el sistema de tiempo continuo, en el que el tiempo puede tomar un valor dentro de un conjunto cerrado y continuo. Por otro lado tenemos el sistema de eventos discretos, en el que el tiempo puede tomar cualquier valor dentro de un conjunto discreto (por lo tanto, no continuo).

Una definición importante en los sistemas temporizados es la sincronía.

Se dice que dos eventos son síncronos cuando tienen el mismo tag.

Por extensión se dice que dos señales son síncronas cuando todos los eventos tienen los mismos tags.

Por extensión se dice que un sistema es síncrono si todas sus señales son síncronas entre sí.

En cambio, una posible definición de sistema no temporizado puede ser la siguiente:

Un sistema se define como no temporizado cuando el conjunto formado por todos los *tags* no está totalmente ordenado, sino que el orden es únicamente parcial.

Imaginemos que tenemos los sistemas mostrados en la figura 2.2 con su línea temporal. Ambos sistemas tienen el mismo comportamiento, por lo que pueden ser descritos por el mismo modelo. No obstante, en dicho modelo, todos los tags no estarán ordenados, como se puede ver en la secuencia de operación; en la que el tag del evento de la señal a puede ser igual o mayor que el tag de la señal b. De la misma forma nos podemos encontrar con comportamientos que no cambiarían si una señal cambiase antes, igual o después que otra.

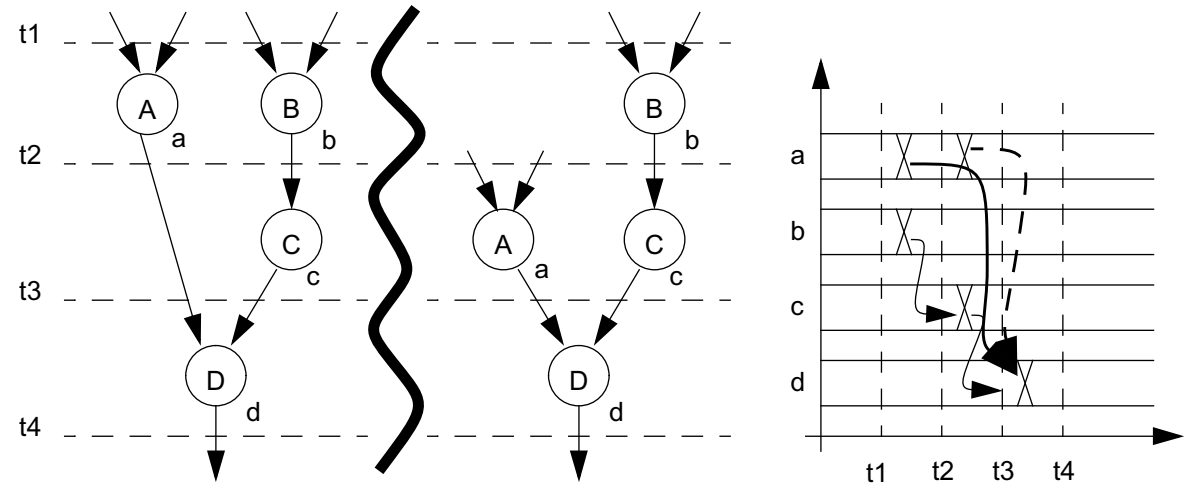


Figura 2.2.- Comportamiento de un sistema no temporizado.

2.3. Modelado del sistema

En el modelado del sistema hay que hacer una distinción entre el lenguaje con el que se describirá o especificará el sistema, y el modelo de computación subyacente. Esto es debido a que un mismo modelo de computación puede ser utilizado con la descripción de diferentes lenguajes, o bien, un lenguaje puede soportar varios modelos. Así por ejemplo, los lenguajes de descripción de hardware VHDL y Verilog pueden soportar entre otros el modelo de computación de eventos discretos y el de máquinas de estado finito.

El modelo de computación afecta a la expresividad, y nos va a indicar que va a ser entendible en el lenguaje utilizado. En este apartado, vamos a tratar de los modelos de computación, y no de los lenguajes.

2.3.1. Modelo de eventos discretos (DE)

El tiempo es una parte integral de este modelo, en el que los eventos suelen llevar asociados marcas temporales totalmente ordenadas. Esta aproximación suele ser muy utilizada para la simulación de sistemas digitales en los que las señales pueden ser fácilmente modeladas de esta forma (mostrada en la figura 2.3). Por lo tanto, y debido a la información de las señales, los procesos únicamente podrán cambiar (y necesitarán ser computados) cuando alguna de las señales que estén involucradas en el proceso en cuestión cambie.

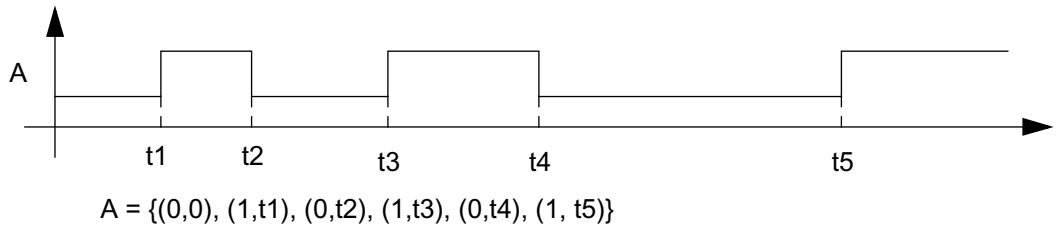


Figura 2.3.- Señal modelada utilizando el modelo DE.

Según el razonamiento anterior, este modelo será más eficaz cuando la actividad sea baja, es decir, para sistemas con secciones frecuentemente ociosas o de operación autónoma (las cuales no producen cambios en las señales de otros procesos). Luego los eventos simultáneos presentan un reto importante en este modelo ya que dejamos de tener marcas temporales totalmente ordenadas, es decir, un sistema temporizado.

Supongamos que tenemos el sistema mostrado en la figura 2.4(a). Según este modelo, los procesos únicamente deben ser evaluados cuando cambian sus entradas siguiendo el ciclo temporal mostrado en la figura, primero se evaluará A, después B y finalmente C.

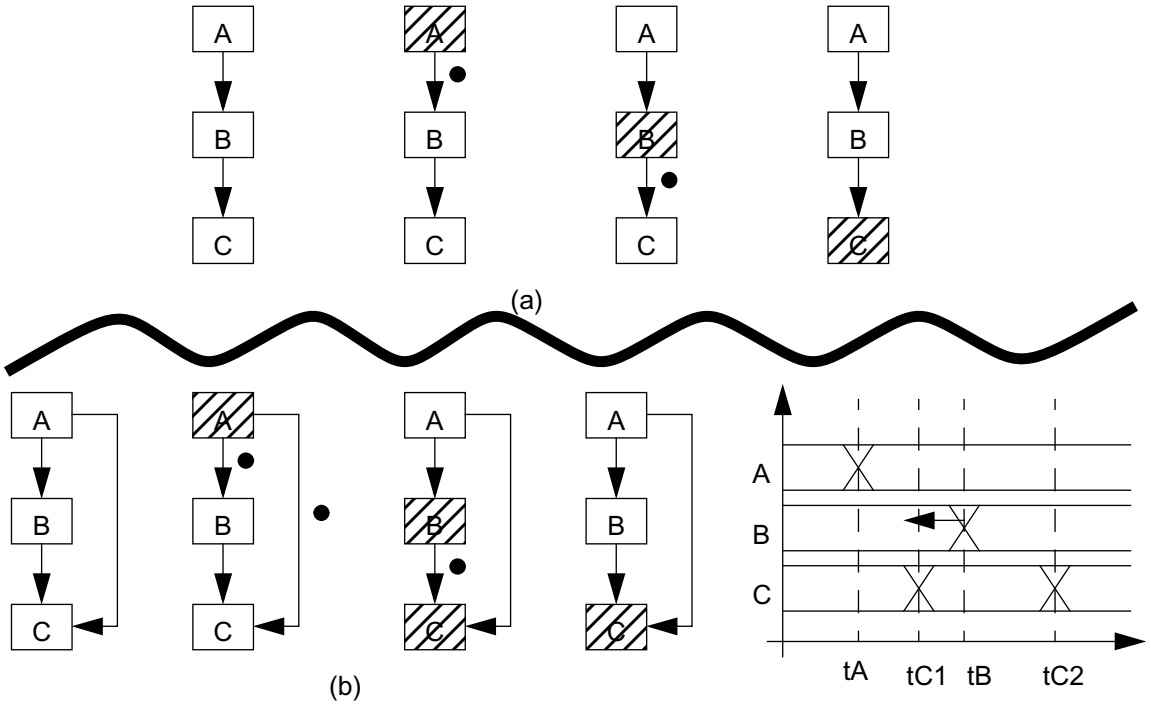


Figura 2.4.- Ejemplos de sistemas utilizando el modelo DE.

Pero supongamos ahora que tenemos el sistema modelado en la figura 2.4(b). Según este modelo, el proceso C será evaluado dos veces: una cuando acaba el proceso A y otra cuando acaba el proceso B. No obstante, el problema consiste en si se dispone del tiempo suficiente para que C se evalúe dos veces, o si mientras se evalúa el primer proceso, ya empieza la segunda evaluación (ha acabado el proceso B). En este último caso (de posibilidad de eventos simultáneos en los procesos C y B), ¿cómo se evalúa C?. La solución a este problema no es estándar y cada simulador la ataca de diferente forma.

Este modelo es muy utilizado ya que existen un elevado número de simuladores basados en él. Entre ellos podemos destacar los simuladores basados en los lenguajes de descripción de hardware VHDL y Verilog, o el simulador Ptolemy.

2.3.2. Modelo de máquinas de estado finito (FSM)

Un modelo de FSM está basado en los autómatas finitos, que tienen una de sus principales utilidades (en el campo hardware) en el diseño de sistemas digitales secuenciales. Este

modelo también está basado en grafos y tiene dos componentes bien diferenciados: estados, que identifican las diferentes situaciones en las que se puede encontrar el sistema (identificados por círculos etiquetados); y transiciones, que identifican los cambios de estados y las condiciones necesarias para que se produzcan (identificados con arcos etiquetados).

La FSM es un modelo muy atractivo para sistemas empotrados, ya que suele ser fácil la obtención de un modelo a partir de sus estados. Una FSM tradicional consiste en un conjunto de entradas, salidas y estados, y una función de salida y de próximo estado, es decir, (I, O, S, δ, λ).

Entre las principales propiedades de las FSM podemos encontrar las siguientes:

- Son buenas para especificar cualquier tipo de control secuencial.
- No forman un modelo completo, desde el punto de vista de Turing.
- Sus dominios típicos de aplicación son tareas de control intensivo y los protocolos.

El gran problema que muestra este modelo es que los sistemas con un elevado grado de concurrencia o grandes necesidades de memoria requerirán un elevado número de estados que generará un modelo poco útil. Para paliar esta problemática se pueden utilizar tres técnicas:

- Jerarquía, con ella sustituiremos una parte completa del diagrama por un solo estado.
- Concurrencia, se permite que varías máquinas estén activas simultáneamente, por lo que varios estados serán representados por uno solo.
- No determinismo, permitiendo situaciones no especificadas o desconocidas, evitaremos la necesidad de añadir estados para modelarlas.

Otro método de reducción de estados es la utilización de una versión extendida de FSM, llamada FSM comunicantes (CFSM) de tal forma que existe una parte FSM (para controlar las acciones) y otra parte de computación de datos (para el procesamiento). Por lo tanto, esta nueva extensión es muy parecida a las máquinas de estado algorítmicas (ASM).

2.3.3. Modelo de redes de proceso de flujo de datos (DF)

El modelo DF es muy útil para aquellos sistemas que son dominados por la operación con los datos.

Una red de DF es una colección de nodos funcionales que están conectados y comunicados a través de una FIFO ilimitada, en la que se van alamacenando los datos de entrada hasta que pueden ser procesados en el orden de llegada.

Este modelo está basado en grafos, en los que podemos distinguir nodos y arcos dirigidos. Los nodos, denominados comúnmente “actores”, realizan la computación; mientras que los arcos muestran la comunicación, así como las etapas de las FIFOs ocupadas.

Supongamos el sistema modelado por el grafo de la figura 2.5. En primer lugar podemos ver la secuencia de operaciones del sistema, utilizando un patrón con dos datos de entrada. Para poder retrasar uno de los sumandos se debe inicializar dicho sumando a un determinado valor inicial. Luego, en primer lugar se multiplica el valor ‘0’ de la señal de entrada por cada

una de las constantes (c1 y c2). Seguidamente estos nuevos valores pasan a las entradas de la operación suma, pero no debemos olvidar que en este momento uno de los sumandos tiene un valor más que el otro debido a la inicialización. Seguidamente se realiza la suma de los sumandos correctos, a la vez que se realizan las multiplicaciones con el valor '1' de la entrada. Para poder realizar estas operaciones cada una de ellas debe tener como entrada una FIFO ilimitada, como se muestra en la figura 2.5(b).

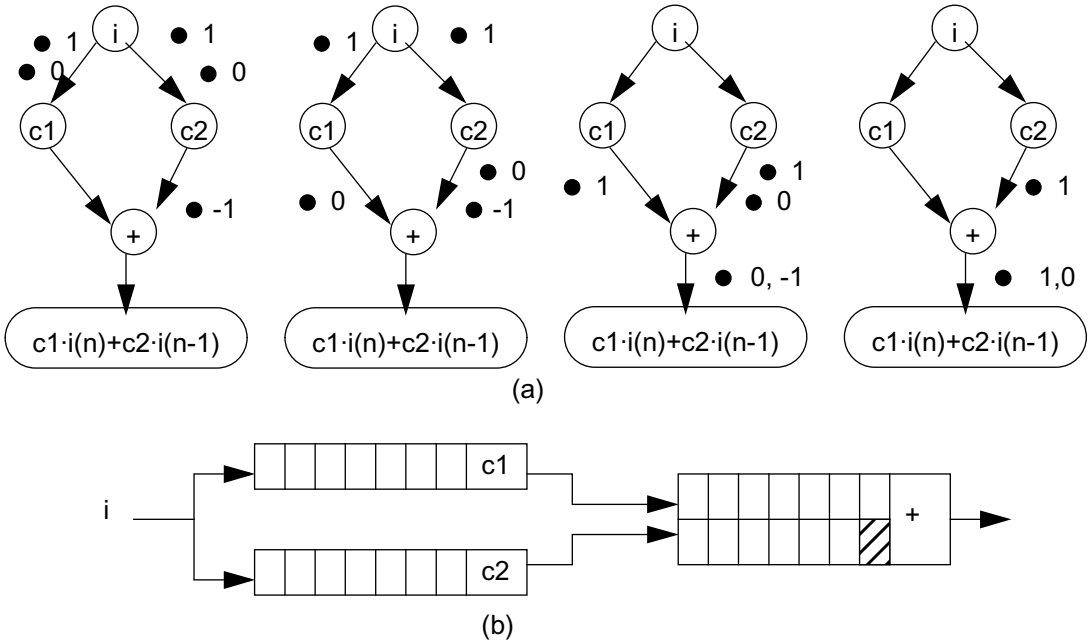


Figura 2.5.- Ejemplo de un sistema modelado según el modelo DF.

No obstante, una pregunta que hay que plantearse es si el orden en el que se ejecutan los diferentes actores afectará al resultado final; situación que no debe permitirse ya que el comportamiento no será el esperado. Por ejemplo en el sistema anterior, si la suma se realiza antes que la multiplicación por la constante c2, desaparece el defase temporal entre las dos ramas y el comportamiento será otro diferente. Por lo tanto, la propiedad clave de este modelo será que la secuencia de salida no deba depender del tiempo de disparo de los actores. Para ello, cada uno de los actores debe ser temporizado para garantizar el comportamiento correcto.

Entre las principales ventajas de las que goza este modelo, podemos destacar las siguientes:

- Fácil de usar
- Algoritmos potentes para verificación y síntesis
- Concurrencia explícita.

Mientras que entre las principales desventajas podemos encontrar las siguientes:

- Síntesis eficaz sólo para modelos restringidos.
- No se puede describir control reactivo.

2.3.4. Modelo de redes de Petri

Las redes de Petri pueden describir explícita y gráficamente secuencialidad/causalidad, conflictos/elecciones no deterministas, y concurrencia; por lo que suele ser utilizado para aplicaciones de computación distribuida, manufactura, control y redes de comunicaciones.

Un modelo de redes de Petri está basado en grafos donde podemos distinguir tres componentes fundamentales:

- lugares, que identifican los posibles estados y condiciones en los que se puede encontrar el sistema (representados por círculos);
- transiciones, que identifican las características que se van a cumplir cuando se puedan producir los cambios de lugares (representadas por una pequeña línea gruesa o una cajita);
- arcos dirigidos, que conectan los diferentes lugares con las respectivas transiciones (representados por flechas finas); y
- tokens, que indican la situación en la red donde se encuentra actualmente el sistema (representados por pequeños puntos negros).

La evolución del sistema modelado con redes de Petri se basa en el cambio de los tokens por los diferentes lugares de la red. El cambio de los tokens se produce cuando una transición sea disparada. Veamos el ejemplo del sistema mostrado en la figura 2.6. En primer lugar se disparan las dos transiciones superiores, por lo que el token pasa a los lugares centrales. Después se dispara la transición inferior, pasando los tokens al último lugar de la red.

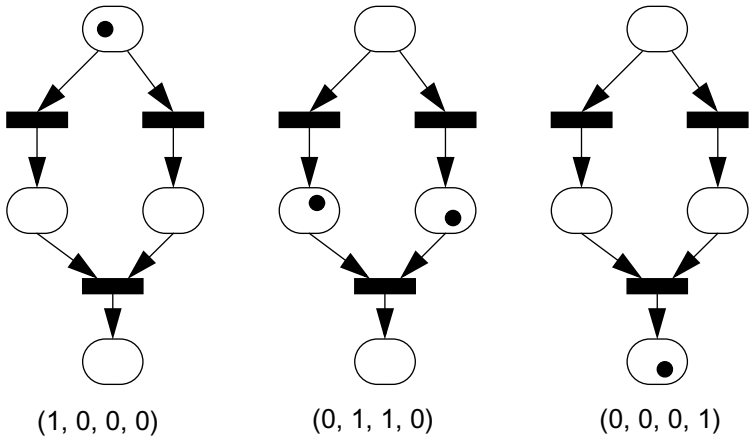


Figura 2.6.- Secuencia de disparos de una red de Petri.

En las redes de Petri se denomina marca a la distribución de los tokens por los diferentes lugares de la red. Una transición podrá ser disparada cuando todos los lugares de entrada tengan los token necesarios (usualmente indicados en el arco). Cuando una transición se dispara, los lugares de entrada pierden los tokens que pasan a los lugares de salida, según la indicación de los arcos (si no hay ninguna indicación, es que el número de tokens involucrados es igual a la unidad).

Para que una red de Petri sea útil, y sobre todo en sistemas hardware, debe cumplir una serie de propiedades:

- Liveness (vida).- cualquier transición debe ser disparable partiendo de cualquier marca. Esta propiedad implica que se puede llegar a la situación inicial, es decir, una red cerrada.
- Boundness.- el número de tokens en cualquier lugar no puede crecer indefinidamente (ya que tendría requerimientos de memoria infinitos).
- Freechoice (libre elección).- el comportamiento depende de las marcas y no del tiempo de llegada (no debe existir ninguna dependencia temporal). Si en la porción mostrada en la figura 2.7, la transición t1 se dispara antes que la t3, los lugares p4 y p5 podrán ser perfectamente alcanzables (situación superior); mientras que si la transición t3 se dispara antes que la t1, el lugar p4 ya no será alcanzable puesto que desaparece el token del lugar p3.

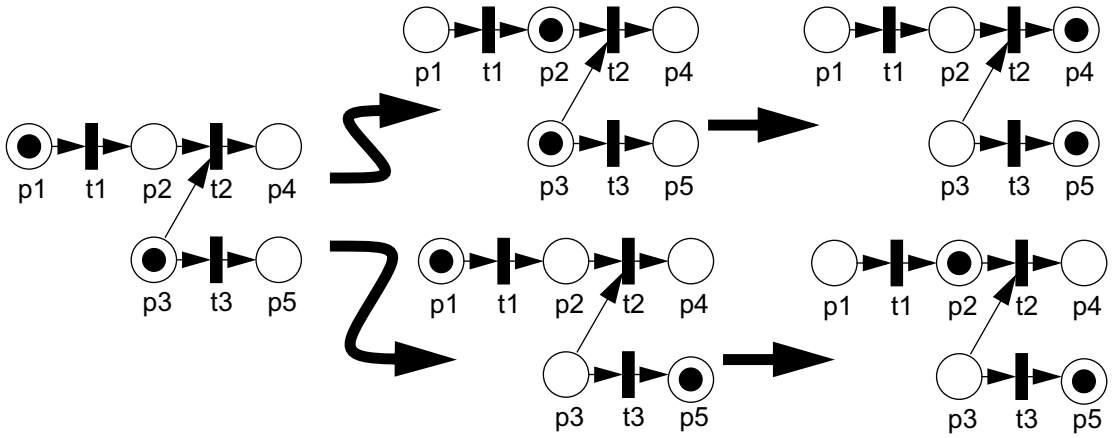


Figura 2.7.- Porción de una red de Petri de no libre elección.

2.4. Modelado de la arquitectura

Para poder implementar un sistema, debemos conocer la arquitectura en la que va a ser alojado. La necesidad de este conocimiento radica en la necesidad de conocer las primitivas (componentes) con los que podemos trabajar, y así, por ejemplo, la partición HW/SW será diferente. De tal forma que las metas de la arquitectura son las siguientes:

- Indicar el número de componentes
- Indicar el tipo de componentes
- Indicar el tipo de conexión entre los diferentes componentes

La principal diferencia en las arquitecturas de sistemas empotrados se basan en el tipo de elemento software, ya que de él va a depender las funciones implementadas vía software. Entre estos elementos podemos encontrar los DSP (Digital Signal Processor), los microcontroladores y los procesadores. No obstante, existen algunos modelos que se prestan mejor para la implementación en una serie de arquitecturas más especificas, como pueden ser las arquitectu-

ras basadas en un camino de datos y el modelo de flujo de datos. A pesar de ello, en este apartado vamos a tratar el tema desde un punto de vista muy genérico ya que los pormenores serán tratados en el siguiente tema.

2.4.1. Arquitecturas basadas en DSP

Un DSP se puede definir de la siguiente forma:

> Un DSP es un dispositivo que ejecuta un algoritmo sobre señales digitales y devuelve la señal digital tratada.

por lo tanto va a realizar un tratamiento digital, generalmente aritmético.

Entre las principales características de las que disponen los DSP de forma común, podemos encontrar las siguientes:

- Algoritmos. Un DSP implementará un conjunto de algoritmos sobre la señal digital. Estos algoritmos caracterizarán a un DSP en concreto, pero no la forma de implementar estos algoritmos; es decir, un DSP podría implementar un algoritmo utilizando un microprocesador y otro DSP podría implementar el mismo algoritmo con un ASIC, y los dos DSP estarían en la misma categoría. Entre los tipos más comunes de algoritmos podemos encontrar codificación y decodificación de voz, encriptación y desencriptación de voz, reconocimiento de voz, síntesis de voz, identificador de voz, compresión y descompresión de imágenes, ...
- Tasa de muestreo. Ésta es una característica clave en un sistema DSP, ya que indica la tasa de muestras que son procesadas, producidas o introducidas en él, siendo un parámetro íntimamente relacionado con la velocidad del sistema, junto con la complejidad del algoritmo. También existen DSP que pueden operar a diferentes tasas de muestreo (DSP multitasa). El rango común de las tasa de muestreo abarca desde las milésimas de hercios hasta las decenas de gigahercios.
- Tasa de reloj. La tasa de reloj se puede definir como la tasa en la que el sistema realiza su unidad más básica de trabajo. Para los sistemas DSP, la razón entre las tasas de reloj y muestreo es una de las características más importantes a la hora de determinar cómo se implementará el sistema, ya que determinará la cantidad de hardware necesario para un sistema empotrado con una determinada complejidad.
- Representación numérica. Las operaciones aritméticas, tales como sumas y multiplicaciones, forman el corazón de los algoritmos implementados en los DSP. Por lo tanto, la representación numérica y el tipo de aritmética utilizada pueden tener una profunda influencia en el comportamiento y las prestaciones del DSP. La elección más importante que deben abordar un diseñador es decidir si elegir una aritmética de punto fijo o de punto flotante.

El rango de aplicación de los DSP abarca una gran variedad de aplicaciones, por lo que no existe un DSP genérico que tenga las características óptimas para todas las aplicaciones. Así existe una división de los DSP en función de la aplicación a la que va dirigido. Esta clasificación divide a las aplicaciones en los siguientes grupos:

- Sistemas empotrados de bajo coste.

- Aplicaciones de altas prestaciones.
- Aplicaciones multimedia basada en PC.

2.4.2. Arquitecturas basadas en microcontroladores

Un controlador es un dispositivo con un conjunto de instrucciones reducido, pensado en la ejecución de un único (o unos pocos) programa. El esquema es muy similar a un sistema procesador exceptuando el sistema de memoria, ya que da un tratamiento diferente a los datos y a las instrucciones. En la figura 2.8 podemos ver el esquema a nivel de bloques de un microcontrolador.

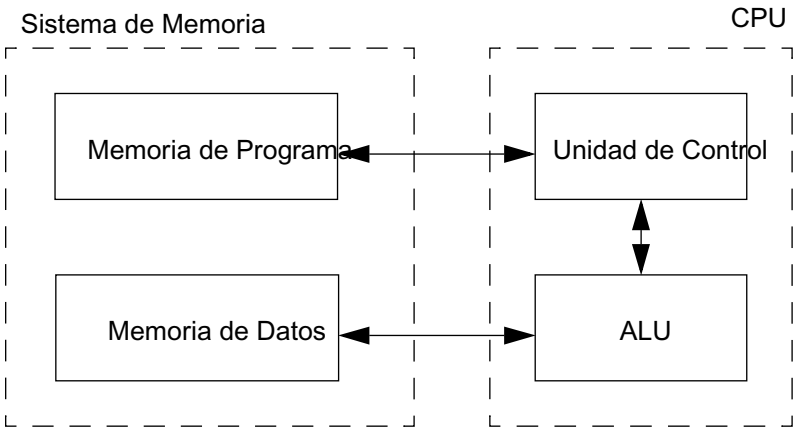


Figura 2.8.- Esquema a nivel de bloques de un microcontrolador.

Esta arquitectura suele ser utilizada cuando los requisitos son los siguientes:

- Alto volumen de producción
- Alta calidad y fiabilidad
- Bajo coste y mantenimiento
- Tiempo de salida al mercado reducido
- Soporte para múltiples versiones del sistema
- Arquitectura seleccionada previamente

2.4.3. Arquitecturas basadas en procesadores

Un procesador es un dispositivo pensado en la ejecución de muchos programas, por lo que se le da el mismo tratamiento a las instrucciones y a los datos. Las principales características de un sistema procesador, y los elementos a considerar para su elección, son las siguientes:

- Conjunto de instrucciones, que nos va a determinar el compilador para poder testear el código de las operaciones implementadas vía software. Además, este conjunto es la frontera entre el hardware y el software (será tratado vía software, pero cada instrucción tiene una equivalencia directa con un recurso hardware).

- Data-path, que es el hardware con el que se van a implementar el conjunto de instrucciones anteriores. La implementación de este elemento va a determinar las prestaciones del sistema completo.
- Sistema de memoria, ya que va a determinar las características del formato, así como las políticas de escritura que se implementarán.

2.4.4. Arquitecturas basadas en componentes

Extraído de la ingeniería clásica, un sistema está formado por la conexión de una serie de componentes, que son subsistemas autocontenidos que pueden ser utilizados como bloques en un sistema mayor. El componente genera un servicio (información) determinado a su entorno a través de unas interfaces bien especificadas. Éste puede tener una estructura interna compleja que puede no ser visible o de interés para el usuario del componente, sino que únicamente necesita verlo como una caja negra que realiza una determinada función. Para que un componente sea útil, debe poder verse desde una serie de puntos de vistas:

- Una unidad de provisión de servicio, es decir, un bloque que realiza una determinada función.
- Una unidad para validación, es decir, el componente debe poder ser validado de forma aislada, para garantizar su correcto funcionamiento en el entorno elegido.
- Una unidad para detección de errores, es decir, el componente debe poder ser observado en su interior para que un fallo en el interior del componente no falsee (sin considerarlo) el resultado final del sistema.
- Una unidad para reuso, es decir, el componente debe poder ser reutilizado para otro tipo de sistemas.
- Una unidad de diseño y mantenimiento, es decir, el componente debe poder ser diseñado y poder disponer de técnicas de mantenimiento para eliminación de errores.

Extrapolando a los sistemas empotrados, un componente será un computador autocontenido con su propio hardware (procesador, memoria, interface de comunicación, interface con el objeto controlado) y software (programas de aplicación, sistema operativo), que realiza un conjunto de funciones bien definidas. En función de la función que realiza, podemos encontrar cuatro tipos de componentes:

- Transductor inteligente. Consiste de un elemento sensor/actuador y un microcontrolador asociado que soporta una interface estándar para un bus de campo.
- Componente interface. Tiene dos interfaces, la interface de red de comunicación a la red de sistema, y la interface del objeto controlado al objeto controlado.
- Componente gateway. Tiene dos interfaces de red de comunicación a la red del sistema, que conecta un cluster a la gateway.
- Componente computacional. Tiene una interface de red de comunicación a la red del sistema y realiza una función computacional del sistema.

Por lo tanto, el sistema final será la conexión de los componentes necesarios para obtener el comportamiento deseado.