Tema VI: Otros métodos de la Ingeniería de Software

Introducción a las características, requerimiento y modelado de sistemas en Tiempo Real (STR).

Introduccion

A medida que los computadores son más pequeños, rápidos, fiables y baratos, su rango de aplicaciones se amplía.
Una de las áreas de expansión más rápida de la explotación de computadores es aquélla que implica aplicaciones cuya función principal no es la de procesar información, pero que precisa dicho proceso de información con el fin de realizar su función principal. Este tipo de aplicaciones de computador se conocen genéricamente como de tiempo real o embebidas.
Se ha estimado que el 99 por ciento de la producción mundial de microprocesadores se utiliza en sistemas embebidos.
Estos sistemas plantean requisitos particulares para los lenguajes de programación.

Una definición mas especifica es "*Cualquier sistema en el que el tiempo en el que se produce la salida es significativo. Esto es generalmente porque la entrada corresponde a algùn movimiento en el mundo físico, y la salida esta relacionada con dicho movimiento. El intervalo entre el tiempo de entrada y el de salida debe ser lo suficientemente para un a temporalidad aceptable*".
O la mencionada por Young (1982) "*Cualquier actividad o sistema de proceso de informaciòn que tiene que responder a un estimulo de entrada generado externamente en periodo finito y especificado",* o incluso la propuesta por Randell (1995), "*Un sistema de tiempo real es aquel a1 que se le solicita que reaccione a estímulos del entorno (incluyendo el paso de tiempo físico) en intervalos del tiempo dictados por el entorno.".*
Estos tipos de sistemas se pueden distinguir de aquéllos otros en los que un fallo al responder puede ser considerado como una respuesta mala o incorrecta.
Por tanto, la correcciòn de un sistema de tiempo real no sólo del resultado lógico de la computaciòn, sino tambièn del tiempo en el que se producen los resultados.

Clasificacion de los STR (temporal)

Las personas que trabajan en el campo del diseño de sistemas de tiempo real distinguen frecuentemente entre sistemas de tiempo real estrictos (hard) y no estrictos (soft)

Los sistemas de tiempo real **estrictos** son aquéllos en los que es absolutamente imperativo que las respuestas se produzcan dentro del tiempo límite especificado.

Los sistemas de tiempo real **no estrictos** son aquéllos en los que los tiempos de respuesta son importantes pero el sistema seguirá funcionando correctamente aunque los tiempos límite no se cumplan ocasionalmente.
Los sistemas no estrictos se pueden distinguir de los interactivos, en los que no hay tiempo límite explícito.
El tiempo límite puede no alcanzarse ocasionalmente (normalmente con un límite superior de fallo en un intervalo definido).
El servicio se puede proporcionar ocasionalmente tarde (de nuevo, con un límite superior en la tardanza).

Ejemplos de STR según su aplicación

1. Control de procesos
2. Fabricaciòn
3. Comunicaciòn mando y control
4. Sistema de computador embebido generalizado

1.- Control de procesos

El primer uso de un computador como componente en un sistema más grande de ingeniería se produjo en la industria de control de procesos (1960). Actualmente, es norma habitual el uso de los microprocesadores.
Consideremos un sistema de control de fluido, ver figura, el ejemplo sencillo, donde el computador realiza una Única actividad: garantizar un flujo estable de líquido en una tubería controlando una válvula. Si se detecta un incremento en el flujo, el computador debe responder alterando el ángulo de la vàlvula; esta respuesta se debe producir en un periodo finito si el equipamiento del final receptor de la tubería no está sobrecargado.
Hay que indicar que la respuesta actual puede implicar una computación compleja en relación con el cálculo del nuevo ángulo de la válvula.

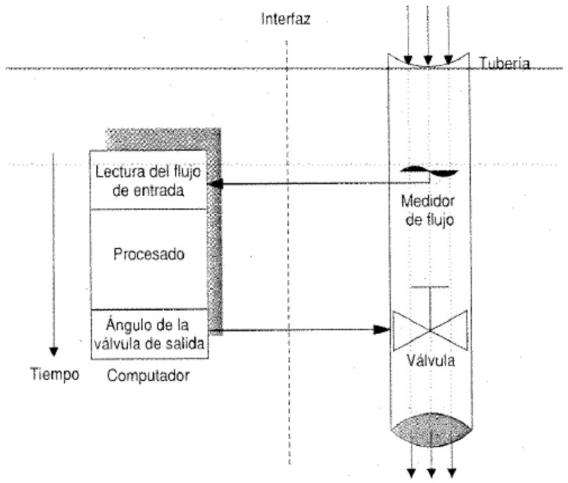

Figura 1.1. Un sistema de control de fluido.

En la figura siguiente se presenta un sistema de control de proceso, es decir, un computador de tiempo real embebido en un entorno completo de control de procesos. El computador interacciona con el equipamiento utilizando sensores y actuadores.
Una válvula es un ejemplo de un actuador, y un transductor de presión o temperatura es un ejemplo de censor. (Un transductor es un dispositivo que genera una señal eléctrica que es proporcional a la cantidad física que esta siendo medida).

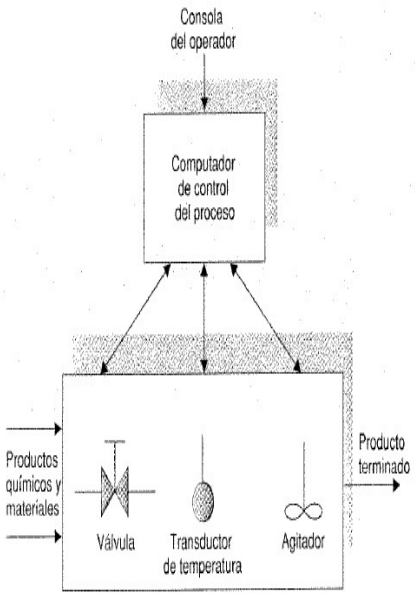

Figura 1.2. Un sistema de control de procesos.

El computador controla la operación de los sensores y actuadores para garantizar que las operaciones correctas de la planta se realizan en los tiempos apropiados. Donde sea necesario se deben insertar convertidores analògicos-digitales entre el proceso controlado y el computador.

2.- Fabricaciòn

El uso de computadores en fabricación permiten:

- mantener bajos los costes de producción
- incrementar la productividad.
- integración del proceso completo de fabricaciòn, desde el diseño a la propia fabricación.

En un sistema de control de producción, ver figura, consta de una variedad de dispositivos mecánicos (como máquinas herramientas, manipuladores y cintas transportadoras), todos los cuales necesitan ser controlados y coordinados con el computador.

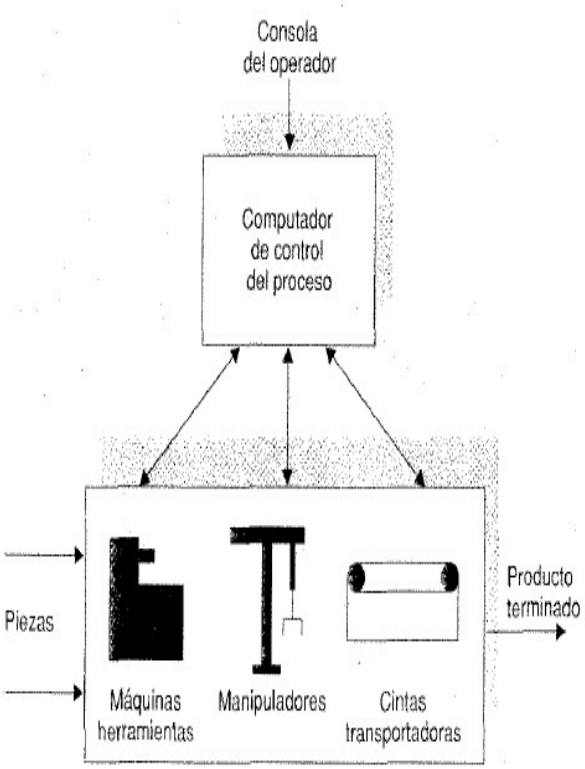

Figura 1.3. Un sistema de control de producción.

3.- Comunicación mando y control

Los sistemas de "comunicación y mando de control", cubren un amplio rango de aplicaciones dispares que exhiben características semejantes; por ejemplo, la reserva de plazas de una compañía aérea, las funcionalidades médicas para cuidado automático del paciente, el control del tráfico aéreo y la contabilidad bancaria remota.
Cada uno de estos sistemas, ver figura, consta de un conjunto complejo de políticas, dispositivos de recogida de información y procedimientos administrativos que permiten apoyar decisiones y proporcionar los medios mediante los cuales puedan ser implementadas.

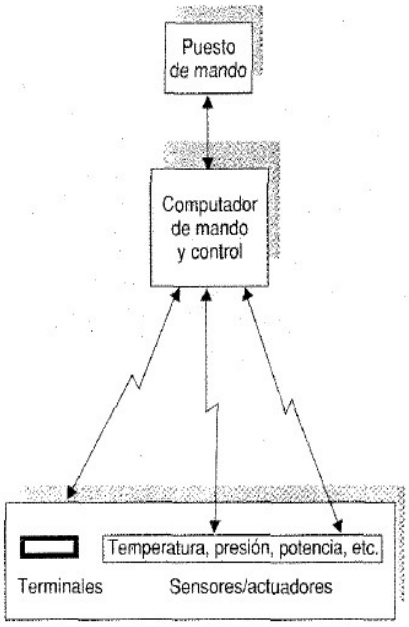

Figura 1.4. Un sistema de mando y control.

Los dispositivos de recogida de información y los instrumentos requeridos para implementar decisiones estarán distribuidos sobre un área geográfica amplia.

4.- Sistema de computador embebido generalizado

En cada uno de los ejemplos mostrados, el computador interacciona directamente con el equipamiento físico en el mundo real.
Con el fin de controlar esos dispositivos del inundo real, el computador necesitará muestrear los dispositivos de medida a intervalos regulares; por lo tanto, se precisa un reloj de tiempo real. Normalmente, existe también una consola de operador para permitir la intervención manual.

El operador humano se mantiene constantemente informado del estado del sistema mediante *display* de varios tipos, incluyendo gráficas.
Los registros de los cambios de estado del sistema se guardan también en una base de datos que puede ser consultada por los operadores, ya sea para situaciones post *mortem* (en el caso de un caída del sistema) o para proporcionar información con propósitos administrativos.
Por supuesto, esta información se está utilizando cada vez más para apoyo en la toma de decisiones en los sistemas que estàn funcionando habitualmente.
Por ejemplo, en las industrias químicas y de procesado, la monitorización de la planta es esencial para maximizar las ventajes económicas, más que para maximizar simplemente la producción.
Las decisiones relativas a la producción en una planta pueden tener repercusiones serias para otras plantas ubicadas en sitios remotos, particularmente cuando los productos de un proceso se están utilizados como materia prima para otro.
El software que controla las operaciones del sistema, ver figura de un sistema embebido típico, puede estar escrito en modulos que reflejan la naturaleza fìsica del entorno.

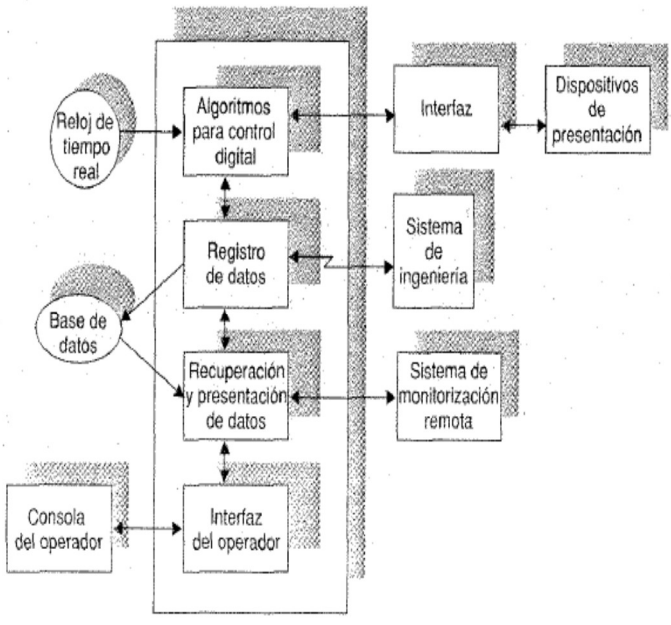

Figura 1.5. Un sistema embebido típico.

Normalmente habra un algoritmo que contenga los algoritmos necesarios para controlar fisicamente los dipositivos, un módulo responsable del registro de los cambios de estado del sistema, un módulo para recuperar y presentar dichos cambios, y un modulo para interaccionar con el operador.

Tareas en tempo real

Las actividades que se realizan en un sistema de tiempo real se las denomina "tareas".
Estas tienen varias tipos de propiedades:

- Funcionales, es decir, que hacen
- Temporales, cuando lo hacen
- Facilidad, seguridad, etc.

El comportamiento temporal de las tareas se especifica mediante sus atributos temporales:

- Cuando se ejecutan: esquema de activación
- Que plazo tienen para ejecutar cada acción

La figura siguiente refiere al esquema de una tarea en tiempo real. En ella se aprecian un momento de "activación" y un momento "limite"

Y el tiempo que transcurre entre ambos momentos es el "plazo de ejecución".
Ademas se puede ver que el "arranque" de la ejecución de la tarea no es precisamente en el momento de activación, sino, ciertamente, en un instante después de este momento.
Ademas, desde el arranque hasta la terminación es donde se "ejecuta la tarea".
Ahora bien, para finalizar, se aprecia también en el grafico que el tiempo de respuesta se inicia en el momento de la activación hasta terminar la ejecución de la tarea.

Caracterìsticas de los STR

- **Grande y complejo**
- **Manipulaciòn de numeros reales**
- **Fiables y seguros**
- **Control concurrente de los distintos componentes separados del sistema**
- **Funcionalidades en tiempo real**
- **Interaciòn con interfases de hardaware**
- **Implementacion eficiente y entorno de ejecuciòn**

Grande y complejo

La variedad es la de las necesidades y actividades en el mundo real y su reflejo en un programa. Pero el mundo real está cambiando continuamente. Está evolucionando.
Tambièn lo hacen. por consiguiente, las necesidades y actividades de la sociedad. Por tanto. los programas grandes, como todos los sistemas complejos, deben evolucionar continuamente.
Los sistemas embebidos deben responder, *por* definición, a eventos del mundo real.
La variedad asociada con estos eventos debe ser atendida; los programas tenderán, por tanto, a exhibir la propiedad indeseable de grandeza.
Inherente a la definición anterior de grandeza es la noción de *cambio continuo.*
El coste de rediseñar y reescribir software para responder al cambio continuo en los requisitos del mundo real es prohibitivo.
Por tanto, los sistemas de tiempo real precisan mantenimiento constante y mejoras durante sus ciclo de vida. Deben ser extensibles.
Aunque los sistemas de tiempo real son a menudo complejos, las características proporcionadas por los lenguajes y entornos de tiempo real permiten que esos sistemas complejos sean divididos en componentes más pequeños que se pueden gestionar de forma efectiva.

Manipulaciòn de Nùmeros Reales

Como se ha indicado anteriormente, muchos sistemas de tiempo real implican el control de alguna actividad de ingeniería.
Vemos en la figura siguiente, un controlador sencillo que la entidad controlada. la planta, tiene un vector de variables de salida, y, que cambian en el tiempo - y(t).

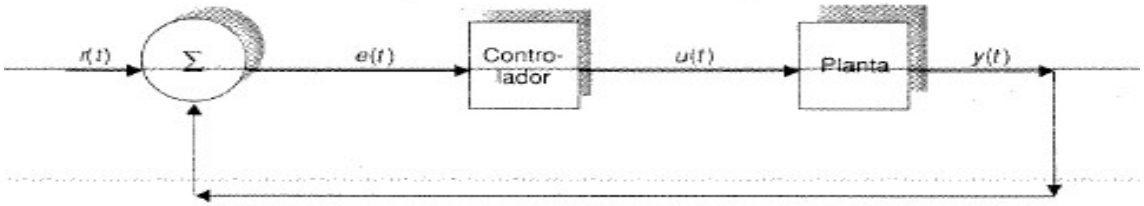

Figura 1.6. Un controlador sencillo.

Estas salidas son comparadas con la señal deseada - ***r(f),*** para producir una señal de error, e(t).
El controlador utiliza este vector de error para cambiar las variables de entrada a la planta, *u(t).* Para un sistema muy sencillo, el controlador puede ser un dispositivo analógico trabajando con una señal continua.
Con el fin de calcular qué cambios deben realizarse en la variables de entrada para que tenga lugar un efecto deseable en el vector de salida, es necesario tener un modelo matemático de la planta.
El objetivo de las distintas disciplinas de la teoría de control es la derivación de estos modelos. Muchas veces, una planta se modela como un conjunto de ecuaciones diferenciales de primer orden.
Éstas enlazan la salida del sistema con el estado interno de la planta y sus variables de entrada. Cambiar la salida del sistema implica resolver estas ecuaciones para conseguir los valores de entrada requerido.
La mayoría de estos sistemas físicos presentan inercia, por lo que el cambio no es instantáneo.
Un requisito de tiempo real para adaptarse ante una nueva consigna en un periodo de tiempo fijo se añadira a la complejidad de las manipulaciones necesarias, tanto en el modelo matematico como en el modelo físico.
El hecho de que, en realidad, las ecuaciones lineales de primer orden sean sólo una aproximación a las características actuales del sistema tambièn presenta complicaciones.
Debido a estas dificultades -la complejidad del modelo y el número de entradas y salidas distintas (pero no independientes)-, la mayoría de los controladores se implementan mediante computadores.
La introducción de un componente digital en el sistema cambia la naturaleza del ciclo de control.

Veamos ahora el mismo controlador sencillo, pero computarizado. Con un computador se pueden resolver las ecuaciones diferenciales mediante técnicas numéricas, aunque los propios algoritmos necesitan ser adaptados para tener en cuenta el hecho de que las salidas de la planta se estén muestreando.

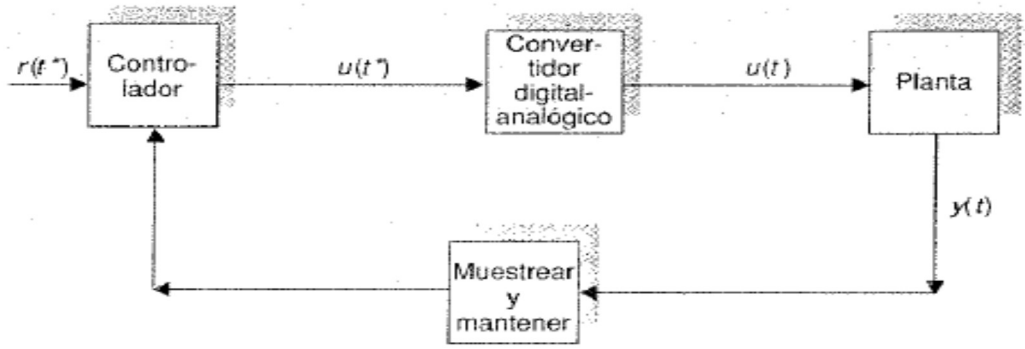

Figura 1.7. Un controlador sencillo computerizado.

Los algoritmos pueden ser matemáticamente complejos y necesitar un grado de precisión elevado.
Un requisito fundamental de un lenguaje de programación de tiempo real es, por tanto, la capacidad para manipular números reales o de coma flotante.

Fiables y Seguros

Cuanto mas entrega la sociedad el control de sus funciones vitales a los computadores, mas necesario es que dichos computadores no fallen.
El Hardware y software de un computador deben ser fiables y seguros. Incluso en entornos hostiles, como en los que se encuentran las aplicaciones militares, debe ser posible diseñar e implementar sistemas que sólo fallen de una forma controlada.
Donde se precise interacción con un operador, deberá tenerse especial cuidado en el diseño de la interfaz, con el fin de minimizar la posibilidad de error humano.
El tamaño y la complejidad de los sistemas de tiempo real exacerban el problema de la fiabilidad no solo deben tenerse en consideración las dificultades esperadas inherentes a la aplicación, sino también aquéllas introducidas por un diseño de software defectuoso.

Control concurrente de los distintos componentes separados del sistema

Un sistema embebido suele constar de computadores y varios elementos coexistentes externos con los que los programas deben interactuar simultáneamente.
La naturaleza de estos elementos externos del mundo real es existir en paralelo.
En nuestro ejemplo típico de computador embebido, el programa debe interaccionar con un sistema de ingeniería (que constará de muchos elementos paralelos. como robots, cintas transportadoras, sensores, actuadores, etc.), y con dispositivos de pantalla del computador (la consola del operador, la base de datos y el reloj de tiempo real).

La velocidad de un computador moderno es tal, que normalmente estas acciones se pueden realizar en secuencia, dando la ilusión de ser simultáneas.
En algunos sistemas embebidos. sin embargo, puede no ser así, como ocurre, por ejemplo, donde los datos son recogidos y procesados en varios lugares distribuidos geográficamente, o donde el tiempo de respuesta de los componentes individuales no se puede satisfacer por un único computador.
En estos casos, es necesario considerar sistemas embebidos distribuidos o multiprocesadores.
Uno de los problemas principales asociados con la produccion de software de sistemas que presentan concurrencia, es cómo expresar la concurrencia en la estructura del programa.
Una posibilidad es dejar todo al programador, que debe construir su sistema de modo que suponga la ejecución cíclica de una secuencia de programa que maneje las distintas tareas concurrentes.

Razones por las cuales se desaconseja:

- Complica la ya difícil tarea del programador ; y le implica a él o ella en consideraciones sobre estructuras que son irrelevantes para el control de las tareas a mano.
- Los programas resultantes seràn màs oscuros y poco elegantes.
- Hace más difícil probar la corrección de programas. y hace mis compleja la descomposición del problema.
- Serà mucho màs difícil conseguir la ejecución del programa en más de un procesador.
- Es màs problemática la ubicaciòn del código que trata con los ellos

Los lenguajes de programaciòn de tiempo real antiguos, por ejemplo RTL/2 y Coral 66, confiaban en el soporte del sistema operativo para la concurrencia; C esta asociado normalmente con Unix o POSIX. Sin embargo. los lenguajes más modernos, como Ada, Java. Pearl y occam, tienen soporte directo para programaciòn concurrente.

Funcionalidades de tiempo real

El tiempo de respuesta es crucial en cualquier sistema embebido. Lamentablemente, es muy dificil diseñar e implementar sistemas que garanticen que la salida apropiada sea generada en los tiempos adecuados bajo todas las condiciones posibles.
Hacer esto y hacer uso de todos los recursos de cómputo todas las veces es imposible.
Por esta razón los sistemas de tiempo real se construyen habitualmente utilizando procesadores con considerable capacidad adicional, garantizando de este modo

que el comportamiento en el peor caso no produzca ningùn retraso inoportuno durante los periodos críticos de operación del sistema.
Dada una adecuada potencia de proceso, se precisa soporte de lenguaje y de ejecución para permitir al programador:

- Especificar los tiempos en los que deben ser realizadas las acciones.
- Especificar los tiempos en los que las acciones deben ser completadas.
- Responder a situaciones en las que no todos los requisitos temporales se pueden satisfacer.
- Responder a situaciones en las que los requisitos temporales cambian dinámicamente (modos de cambio).

Estos requisitos se llaman funcionalidades de control en tiempo real. Permiten que el programa se sincronice con el tiempo.

Por ejemplo, con algoritmos de control digital directo es necesario muestrar las lecturas de los sensores en determinados periodos del día (por ejemplo a las dos de la tarde, a las tres de la tarde, y así sucesivamente), o a intervalos regu1are (por ejemplo, cada 5 segundos) con los convertidores analógico-digitales, las tasas de muestreo pueden variar desde unos pocos hertzios a varios cientos de megahertzios. Como resultado de estas lecturas. se precisarà realizar otras acciones.

Interacción con interfaces hardware

La naturaleza de los sistemas embebidos requiere componentes de computador para interaccionar con el mundo externo.
Eso se necesita para monitorizar sensores y controlar actuadores para un amplia variedad de dispositivos de tiempo real.
Estos dispositivos interactùan con el computador mediante los registros de entrada y salida, y sus requisitos operativos son dependientes de dispositivo y computador.
Los dispositivos pueden generar también interrupciones para indicar al procesador que se han realizado ciertas operaciones o que se han alcanzado ciertas condiciones de error.
En el pasado, la interacción con los dispositivos o bien se realizaba bajo control del sistema operativo, o bien requería que el programador de aplicaciones recurriera a inserciones en el lenguaje ensamblador para controlar y manipular los registros e interrupciones.
Actualmente, debido a la variedad de dispositivos y a la naturaleza de tiempo crítico de sus interacciones asociadas, su control debe ser directo, y no a través de una capa de funciones del sistema operativo.

Además, los requisitos de calidad son argumentos contra el uso de técnicas de programación de bajo nivel.

Implementaciòn eficiente y entorno de ejecución

Puesto que los sistemas de tiempo real son críticos respecto al tiempo, la eficiencia de la implementación serà màs importante que en otros sistemas.
Es interesante que uno de los principales beneficios de utilizar un lenguaje de alto nivel es que permite al programador abstraerse de los detalles de implementación y concentrarse en la resolución del problema.
Debe preocuparse del coste de utilizar las caracteristicas de un lenguaje particular (Ej.: si la respuesta debe ser en microsengundo, no serà necesario utilizar un lenguaje con respuesta en milesegundo).

Diseños de STR

La etapa más importante del desarrollo de cualquier sistema de tiempo real es la generación de un diseño consistente que satisfaga una especificación acreditada de los requisitos.
La disciplina de la ingeniería del software se encuentra ampliamente aceptada como el nucleo del desarrollo de mètodos, herramientas y técnicas con el fin de asegurar la correcta gestión del proceso de producción de software y la consecución de programas correctos y fiables.
Aunque la mayorìa de las aproximaciones al diseño son descendentes, se fundamentan en la comprensión de qué es realizable en los niveles inferiores. En esencia, todos los métodos de diseño incluyen una secuencia de transformaciones desde el estado de los requisitos iniciales hasta el código ejecutable.
Una visión general de los pasos a realizar en el diseño de STR:

1. Especificaciòn de requisitos
2. Diseño arquitectonico
3. Diseño detallado
4. Implementaciòn
5. Prueba
6. Prototipado
7. Diseño de la interfaz hombre – maquina
8. Criterios para la evaluaciòn de los lenguajes de implementacion

Niveles de la especificación de los requerimientos

Hay varios modos de clasificar las formas de notación. McDermid (1989) proporciona una útil descomposición pasa nuestros fines y destaca tres técnicas:

- Informal
- Estructurada
- Formal

Los mètodos informales suelen hacer uso del lenguaje natural y de diversos tipos de diagramas imprecisos.
Los métodos estructurados suelen emplear notación gráfica bien definidos. Dichos esquemas se construyen a partir de un pequeño número de componente, predefinidas que pueden conectarse de una forma controlada. La forma gráfica puede tener también una reprecentación sintáctica en algún lenguaje bien definido.
A pesar de que pueden idearse métodos estructurados muy rigurosos, no pueden, en sí, ser analizados o manipuladas. Para poder efectuar este tipo de operaciones, la notación debe tener una base matemática. Los métodos que presentan dichas propiedades matemáticas se conocen como métodos formales. Éstos tienen la ventaja evidente de que mediante estas notaciones es posible realizar descripciones precisas.

1.- Especificaciòn de requisitos

Es en esta etapa donde se define la funcionalidad del sistema. En cuanto a los factores concretos de tiempo real, deberá hacerse suficientemente explícito el comportamiento temporal del sistema, así como los requisitos de fiabilidad y el comportamiento deseado del sistema en caso de fallos en los componentes.
La fase de requisitos deberá definir también los test de aceptación que se aplicarán al software.
A parte del sistema en sí, es necesario construir un modelo del entorno de la aplicación. Es característico de los sistemas de tiempo real presentar interacciones importantes con su entorno.
Por tanto, cuestiones como la tasa máxima de interrupciones, el número máximo de objetos externos dinámicos (por ejemplo, las aeronaves en un sistema de control de tráfico aéreo) y los modos de fallos, son importantes.

Tècnicas de requisitos y notaciones estructuradas

- PSL (Teichrowy Hershey, 1977)
- CORE (Mullery, 1979), por sus claras ventajas a la hora de obtener un conjunto de'requisitos no ambiguos.
- Además, los métodos orientados al objeto se están haciendo muy populares (Monarchi y Puhr, 1992), y existe un estándar industrial, UML.
- Se han realizado esfuerzos en la línea de los métodos formales para el análisis de requisitos (el proyecto FOREST (Goldsack y Finkelstein, 1991),

que define un esquema lógico para tratar con los requisitos y un método para la elicitación de requisitos).

- Más recientemente, en el proyecto Esprit II ICARUS se ha desarrollado el lenguaje ALBERT (Lenguaje Orientado a Agentes para la Construcción y Elicitación de Requisitos para Sistemas de Tiempo Real; Agent-oriented Langucrge for Buildiizg nnd Eliciting Requirerizent for real-Tinze systems) (Dubois et al., 1995; Bois, 1995).
- VDM (Jones, 1986). Otra técnica que ha conseguido gran apoyo es Z (Spivey, 1989). Ambos métodos emplean la teoría de conjuntos y la lógica de predicados, y representan una mejora considerable sobre las técnicas informales y las meramente estructuradas.

2.- Actividades de Diseño

El diseño de un sistema embebido grande no se puede hacer de una vez. Debe estructurarse de alguna manera.
Para gestionar el desarrollo de los sistemas de tiempo real complejos, se suelen emplear dos aproximaciones complementarias.
La descomposición, como sugiere su nombre, implica una participación sistemática del sistema complejo en partes más pequeñas, hasta poder aislar componentes que puedan ser comprendidos y diseñados por individuos o grupos pequeños.
En cada nivel de descomposición, deberá haber un nivel de descripción apropiado y un método para documentar (expresar) esta descripción.
La abstracción permitirá posponer cualquier consideración referente a los detalles, particularmente las concernientes a la implementación. Esto permite tener una visión simplificada del sistema y de los objetos contenidos en el que, aun así, contendrà las propiedades y características esenciales.
La utilización de la abstracción y la descomposición impregna todo el proceso de ingeniería, y ha influido en el diseño de los lenguajes de programación de tiempo real y en los métodos de diseño de software asociados.
Si se emplea una notación formal para la especificación de requisitos, los diseños de alto nivel deberían utilizar la misma notación, para así poder demostrar si cumplen dicha especificación.
Sin embargo, muchas notaciones estructuradas pretenden servir para completar el diseño de alto nivel, o incluso para remplazar la notación formal. De cualquier forma, un diseño estructurado de alto nivel debería ser una especificación acreditada de los requisitos.

Encapsulamiento

El desarrollo jerárquico del software conduce a la especificación y subsiguiente desarrollo de los subcomponentes del programa. Por la naturaleza de la

abstracción, estos subcomponentes deberan tener roles bien definidos, e interfaces conexiones claras e inequívocas. Cuando la especificación completa del sistema software puede verificarse teniendo en cuenta únicamente la especificaciòn de los subcomponentes inmediatos, se dice que la descomposiciòn es composicional.
Los programas secuenciales son idóneos para los métodos composicionales, y existen técnicas para encapsular y representar subcomponentes:

- Simula introdujo la significativa construcción de clases
- Modula-2 emplea la estructura de modulo que, a pesar de ser menos potente, sigue siendo importante.
- Los lenguajes *orientados al abjeto,* como C++, Java y Eiffel, partiendo de la construcción clase.
- Ada emplea una combinación de módulos y extensiones de tipo para soportar la programación orientada al objeto.

Los objetos, si bien proporcionan una interfaz abstracta, requieren elementos adicionales si han de ser usados en un entorno concurrente.
Habitualmente, esto conlleva la inclusión de alguna noción de proceso.
Tanto la abstracción de objeto como la abstracción de proceso son importantes en el diseño e implementación de los sistemas embebidos fiables.

Cohesión y acoplamiento

Las dos formas anteriores de encapsulamiento conducen al empleo de módulos con interfaces bien definidas (y abstractas).
Pero, ¿cómo debería descomponerse en módulos un sistema grande?
La cohesión y el acoplamiento son dos medidas que describen la vinculación entre módulos.
La cohesión expresa el grado de unión de un módulo: su fuerza interna. Allworth y Zobel (1987) indican seis medidas de cohesión que van desde la más débil a la más fuerte:

- Casual: los elementos del módulo mantienen tan solo vínculos muy superficiales; por ejemplo, el haber sido escritos en el mismo mes.
- Lógica: los elementos del módulo están relacionados desde el punto de vista del sistema completo, pero no en términos reales de software; por ejemplo, todos los gestores de dispositivos de salida.
- Temporal: los elementos del módulo se ejecutan en momentos similares; por ejemplo, las rutinas de arranque.
- Procedural: los elementos del módulo se emplean en la misma sección del programa; por ejemplo, los componentes de la interfaz de usuario.

- De comunicación *(sic):* los elementos del módulo operan sobre la misma estructura de datos.
- Funcional: los elementos del módulo operan conjuntamente para contribuir a la ejecución de una única funciòn del sistema.

El acoplamiento es una medida de la interdependencia entre dos módulos de un programa.
Si dos módulos intercambian información de control entre ellos, se dice que poseen un acoplamiento alto (rígido, o fuerte).
El acoplamiento es débil si únicamente se intercambian datos.
Otra forma de contemplar el acoplamiento es considerar la facilidad con que puede eliminarse un módulo (de un sistema ya completo) y remplazarlo con un módulo alternativo.
En cualquier método de diseño, una buena descomposición es aquélla que posee fuerte cohesión y débil acoplamiento.
Este principio es igualmente válido tanto para el dominio de la programación secuencia1 como para el de la programación concurrente.

Aproximaciones formales

El empleo de redes sitio-transición (Place-Transition) (Brauer, 1980) para el modelado de sistemas concurrentes fue propuesto por C. A. Petri hace más de treinta años.
Estas redes se construyen como un grafo bipartito dirigido con dos tipos de nodos: elementos S, que denotan estados atómicos locales, y elementos T que denotan transiciones. Los arcos del grafo proporcionan la relación entre los elementos S y T. El marcado del grafo se indica mediante marcas sobre los elementos S; el desplazamiento de un testigo representa un cambio en el estado del programa. Mediante reglas se especifica cuándo y cómo se puede mover u testigo de un elemento S a otro a través de un elemento de transición. Las redes de Petri se definen matemáticamente, y son susceptibles de un análisis formal.

Las redes sitio-transición poseen las útiles cualidades de ser simples, abstractas y gráficas, y proporcionan un marco de trabajo general para analizar muchos tipos de sistemas distribuidos y concurrentes.
En contrapartida, pueden dar lugar a representaciones muy grandes y complicadas. Para poder trabajar con un modelo más conciso de tales sistemas se han confeccionado las redes predicado-transición.
Con estas redes, un elemento S puede modelar varios elementos S normales (de igual modo elementos S), y las marcas, que en origen no tenían estructura interna, pueden ser «coloreadas» por tuplas de datos.
Una alternativa a las redes de Petri es el uso de autómatas temporizados y comprobación de modelos (Model Checking). De nuevo, el sistema concurrente

se representa mediante un conjunto de máquinas de estado finito, y se emplean técnicas de exploración de estados para investigar los posibles comportamientos del sistema.

La notación *Conznzunicaring Seqilerztitrl Processes* (CSP: procesos secuenciales que se comunican) fue desarrollada para permitir la especificación y el análisis de sistemas concurrentes.
En CSP, cada proceso se describe en términos de eventos externos, esto es, de acuerdo con la comunicación que mantiene con otros procesos. La historia de un proceso se representa mediante una *traza,* que es una secuencia finita de eventos.
Un sistema representado en CSP puede ser analizado para determinar su comportamiento. En particular, podrán examinarse las propiedades de *seguridad* (safety) y *vivacitiacl* (liveness). Owicki y Lamport (1982) caracterizaron estos dos conceptos como:

- Seguridad: «algo malo no va a ocurrir.
- Vivacidad: «algo bueno va a ocurrir».

Las técnicas de comprobaciòn de modelos pueden usarse también para verificar las propiedades de seguridad y vivacidad.

Metodologia de diseño

Ya se comentó que la mayoría de los diseñadores de sistemas de tiempo real promueven un proceso de abstracción de objetos, y que existen técnicas formales que permiten especificar y analizar sistemas concurrentes con restricciones temporales. No cabe duda de que estas técnicas no estan aùn suficientemente maduras para constituir mètodos de diseño probados y contrastados.
Más bien, la industria de tiempo real usa, en el mejor de los casos, métodos estructurados y aproximaciones de ingeniería de software aplicables a la mayoría de los sistemas de procesamiento de información.
Estos métodos no proporcionan un soporte específico para el dominio de tiempo real; y carecen -de-la riqueza necesaria para explotar completamente la potencia ,de los lenguajes de implementación.
Métodos de diseño estructurados orientados a los sistemas de tiempo real: MASCOT, JSD, Yourdon, MOON, OOD, RTSA, HOOD, DARTS, ADARTS, CODARTS, EPOS, MCSE, PAMELA, HRT-HOOD, Octopiis, etc.

Gomaa (1994) sugiere cuatro objetivos importantes para un método de diseño de tiempo real. Debe ser capaz de:

- Estructurar un sistema en tareas concurrentes.
- Dar soporte al desarrollo de componentes reusables mediante la ocultación de información.

- Definir los aspectos del comportamiento mediante máquinas de estado finito.
- Analizar Las prestaciones de un diseño para determinar sus propiedades de tiempo real.

Ninguno de los métodos anteriores cumple todos estos objetivos.

- RTSA es débil en la estructuración de tareas y ocultación de información.
- JSD no da soporte a la ocultación de información o a las máquinas de estado finito.

Además, pocos de estos métodos soportan directamente las abstracciones habituales del tiempo real estricto (tales como actividades periódicas y esporádicas) que se encuentran en la mayoría de los sistemas de tiempo real estrictos.
Incluso, sólo una minoría impone un modelo computacional que asegure que pueda realizarse un análisis de la temporización efectiva del sistema final. Consecuentemente, su utilización es propensa a errores, y puede conducir a sistemas cuyas propiedades de tiempo real no puedan ser analizadas.

- PAMELA permite la representación de diagramas de actividades cíclicas, máquinas de estado y manejadores de interrupciones. Sin embargo, la notación no está respaldada por abstracciones para los recursos, y en consecuencia los diseños no son necesariamente adecuados para el análisis de temporización.
- HRT-HOOD da soporte a tres de los cuatro requisitos, careciendo, sin embargo, de la posibilidad de definir los aspectos de comportamiento mediante máquinas de estado finito. También, al ser sólo basado en objetos, sus mecanismos de apoyo a la reutilización de componentes es débil.
- HOOD V4 intenta resolver las debilidades de HOOD V3, e incorpora los puntos fuertes de HRT-HOOD. Sin embargo, su soporte para el diseño arquitectónico reusable es débil, y el análisis de prestaciones del diseño no está garantizado.
- EPOS (Lauber, 1989) es otro método que da soporte al ciclo de vida completo de un sistema de tiempo real. Proporciona tres lenguajes de especificación: uno para describir los requisitos del cliente, otro para describir la especificación del sistema, y otro para describir la gestión del proyecto, la gestión de la configuración y las garantías de calidad. Como HOOD y HRT-HOOD, la especificación del sistema tiene una representación gráfica y textual. Pueden representarse actividades periódicas y esporádicas (aunque no son directamente visibles cuando se

manipulan los diagramas), y pueden especificarse los requisitos de temporización. Además, dentro del contexto de EPOS (Lauber, 1989) se aborda el reconocimiento temprano del comportamiento de las aplicaciones de tiempo real: sin embargo. este trabajo se basa en una animación de la especificaciòn del sistema, y no es un anàlisis de temporizaciòn.

Como se comentó en la introducciòn, la mayoría de los métodos tradicionales de desarrollo de software incorporan un modelo de ciclo de vida donde se ubican las siguientes actividades:

- Especificaciòn de requisitos: durante la cual se produce una especificación acreditada del comportamiento funcional y no funcional del sistema.
- Diseño arquitectónico: durante el cual se desarrolla una descripción de alto nivel del sistema propuesto.
- Diseño detallado: durante el cual se especifica el diseño completo del sistema.
- Codificación: durante la cual se implementa el sistema.
- Prueba: durante la cual se comprueba la eficacia del sistema.

Para los sistemas de tiempo real estrictos, se presenta la desventaja importante de que los problema de temporizaciòn sólo se podrán detectar durante la prueba, o, lo que es incluso peor, tras el despliegue (deployment) de la aplicación.
Como es normal, un método de diseño estructurado empleará un diagrama en el que las flechas etiquetadas muestran el flujo de datos a través del sistema, y ciertos nodos representan lugares donde se transforman dichos datos (esto es, los procesos).

- **JSD**. (Jackson, 1975) Emplea una notación precisa para la especificación (diseño de alto nivel) y la implementaciòn (diseño detallado). La implementación no es sólo una reescritura detallada de la especificación, sino el resultado de la aplicación de ciertas transformaciones sobre la especificación inicial con el objeto de incrementar la eficiencia. Un grafo JSD consta de procesos y una red de interconexión. Son de tres clases:
  - Procesos de entrada, que detectan acciones en el entorno y las transfieren al sistema.
  - Procesos de salida, que transfieren al entorno las respuestas del sistema.
  - Procesos internos.

- **Mascot 3 .** Mientras que JSD sólo ha sido utilizado recientemente en el dominio de tiempo real, Mascot se diseñó específicamente para el diseño, la construcción y la ejecución de software de tiempo real.Mascot I (1970).

Fue mejorado Mascot II, posteriormente, en (1980) Mascot III.Masco 3 se caracteriza por emplear gráficos de redes de flujo de datos y un diseño jerárquico.La modularidad es un elemento clave de este método. en el que se pueden identificar módulos para el diseño, la construcción, la implementación y la prueba. Un diseño en Mascot 3 puede expresarse tanto en forma textual como gráfica.Con Mascot2, los algoritmos se codifican en un lenguaje secuencial, como Coral 66 o RTL2 (aunque también se han empleado FORTRAN, Pascal, C y Algol 60) y después este software se aloja en una plataforma de ejecución Mascot de tiempo real.

- **HRT-HOOD.** (Burns y Wellings, 1995) difiere de Mascot y JSD en que aborda directamente las cuestiones de los sistemas de tiempo real estrictos. Se ve el proceso de diseño como una progresión de *compromisos especificos* crecientes (Burns y Lister, 1991). Estos compromisos definen propiedades del diseño del sistema que los diseñadores de niveles más detallados no pueden cambiar.Aquellos aspectos de un diseño sobre los que no se ha contraído ningún compromiso en ningún nivel concreto de la jerarquía son objeto de *ob1igaciones* que deberán resolver los niveles inferiores del diseño. Ya en las etapas tempranas del diseño habrà compromisos sobre la estructura arquitectónica del sistema, en términos de definiciones y relaciones entre objetos. Sin embargo, el comportamiento detallado de los objetos definidos es objeto de las obligaciones que deberán cumplimentarse en un diseño más detallado y en la implementación.
- **UML**. Los principales aspectos de UML que se adecuan al modelado de sistemas embebidos de tiempo real son (Douglass, 1999):

    - Un modelo de objetos (que incorpora atributos para datos, estado, comportamiento, identidad y responsabilidad) que permite capturar la estructura del sistema.
    - Escenarios de casos de uso, que permiten identifica respuestas clave del sistema o las entradas del usuario.
    - Modelado del comportamiento, con máquinas de estado finito (diagrama de estados*)* que facilitan el modelado de la dinámica del comportamiento del sistema.
    - Empaquetado, que proporciona mecanismos para organizar los elementos del modelado; representaciones para la concurrencia, la comunicación y la sincronización (para modelar entidades del mundo real).
    - Modelos de la topología física, donde se emplean diagramas de despliegue para mostrar cómo está compuesto el sistema a partir de dispositivos y procesadores.

- Soporte para patrones y marcos orientados al objeto, que permiten representar soluciones comunes a problemas comunes.

Sin embargo, y como su nombre indica, UML ofrece un lenguaje, no un método. En consecuencia, es preciso aumentar la notación con un proceso de diseño. Douglass (1999) propone ROPES *(Rnpid Object-oriented Process for Embcdtled Systems;* proceso **rápido orientado al objeto para sistemas embebidos) con UML.**

Implementaciòn

Entre la especificación de requisitos de alto nivel y el código de máquina en funcionamiento encontramos un gran hueco, ocupado por el lenguaje de programación.
El diseño de lenguajes es aún un tema de investigación muy activo. Aunque el diseño de los sistemas debería conducir de modo natural hacia la implementación, el poder expresivo de la mayoría de los lenguajes modernos está aún lejos de alcanzarse con las metodologías de diseño actuales.
Sólo comprendiendo qué es posible en la etapa de implementación, podremos conseguir una aproximación apropiada al diseño. Podemos identificar tres clases de lenguajes de programación:

- los lenguajes de ensamblado,
- los lenguajes de implementación de sistemas secuenciales;
- y los lenguajes concurrentes de alto nivel.

Ensamblador

Inicialmente, la mayoría de los sistemas de tiempo real se programaron en el lenguaje de ensamblado (o ensamblador) del computador embebido.
La causa era que la mayoría de los lenguajes de alto nivel no disponían de soporte suficiente para la mayoría de los microcomputadores, y el lenguaje de programación de ensamblado parecía ser la única forma de obtener una implementación eficiente que pudiera acceder a los recursos de hardware.
El principal inconveniente de los lenguajes de ensamblado es que están orientados a la máquina en lugar de al problema. El programador debe preocuparse de problemas que nada tienen que ver con los algoritmos en sí, de manera que éstos se vuelven irreconocibles. Esto hace que los costes de desarrollo se eleven y que resulte difícil modificar los programas cuando le encuentran errores, o cuando es preciso efectuar mejoras.
Los programas escritos en lenguajes ensambladores no es posible llevarlos de una máquina a otra.
Se hace necesario volver a formar a los programadores si se necesita trabajar con otras máquinas diferentes.

Lenguajes de implementación de sistemas secuenciales

A medida que los computadores fueron siendo más potentes y los lenguajes de programación más maduros, y progresaba la tecnología de los compiladores, las ventajas de escribir software de tiempo real en un lenguaje de alto nivel sobrepasaron a los inconvenientes.
Para tratar con las deficiencias de lenguajes como FORTRAN, se desarrollaron nuevos lenguajes específicos para la programación embebida. (Fuerzas Aéreas de EE.UU, se trabajó con Jovial; Reino Unido, se adhirió a Coral 66; grandes corporaciones industriales, como ICI, se adhirieron a RTLJ2).
Más recientemente, han ido ganando popularidad los lenguajes de programación C y C++.
Todos estos lenguajes tienen un punto en común: son secuenciales. También suelen presentar carencias en lo que respecta a sus funciones de control y fiabilidad para tiempo real. En consecuencia, suele ser necesario basarse en el soporte del sistema operativo y fragmentos de código en ensamblador.

Lenguajes de programación concurrente de alto nivel

La **computación concurrente** es la simultaneidad en la ejecución de múltiples tareas interactivas.
Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa. Las tareas se pueden ejecutar en una sola unidad central de proceso (multiprogramación), en varios procesadores o en una red de computadores distribuidos.
La programación concurrente está relacionada con la programación paralela, pero enfatiza más la interacción entre tareas. Así, la correcta secuencia de interacciones o comunicaciones entre los procesos y el acceso coordinado de recursos que se comparten por todos los procesos o tareas son las claves de esta disciplina.
Los pioneros en este campo fueron Dijkstra, Hoare, Hanssen, etc.

Pese al creciente empleo de lenguajes orientados a aplicaciones y a los datos o científicas (COBOL y FORTRAN), la producción de software fue haciéndose progresivamente más difícil (1970), a medida que los sistemas informáticos se volvieron más grandes y sofisticados.
Búsqueda por parte del Departamento de Defensa Americano (DoD) de un lenguaje de programación de alto nivel común para todas sus aplicaciones.
A medida que comenzaban a caer los precios del hardware durante los años 1970, el DoD observó los crecientes costes de su software envebido.
Se estimó que en 1973 se habían gastado tres mil millones de dólares tan sólo en software.

Una encuesta que tenía como objeto los lenguajes de programación mostró que en la construcción de aplicaciones embebidas se utilizaban más de 450 lenguajes de propósito general y dialectos incompatibles.
En 1976 se evaluaron los lenguajes existentes en base a un conjunto de requisitos de creciente importancia.
A partir de estas evaluaciones, se extrajeron cuatro conclusiones principales (Whitaker, 1978):

- Ningún lenguaje del momento era apropiado.
- El objetivo deseable era obtener un único lenguaje.
- El estado actual del arte del diseño de lenguajes no servía para cumplir los requisitos.
- Este desarrollo debería arrancar de un lenguaje base apropiado; entre los que se recomendó Pascal, PLl y Algo168.

El resultado fue el nacimiento, en 1983, de un nuevo lenguaje, llamado Ada. En 1995, se modernizó el lenguaje para reflejar los últimos 10 años de experiencia y avances del diseño moderno de lenguajes de programación.

- Modula-1 (Wirth, 1977b), que fue desarrollado por Wirth para su empleo en dispositivos programables y que pretendió.
- Lenguaje C (Kemighan y Ritchie, 1978).
- Modula 2. (Wirth, 1983). un lenguaje para implementación de sistemas de propósito general.
- El lenguaje C se ha desdoblado: C++, que soporta directamente la programación orientada al objeto.
- PEARL, empleado extensamente para aplicaciones de control de procesos en Alemania; Mesa (Xerox Corporation, 1985), utilizado por Xerox e.n . su equipamiento de automatización de oficinas:
- CHILL (CCITT, 1980). desarrollado para la programación de aplicaciones de telecomunicaciones, en respuesta a los requisitos de CCITT. (versión Basic para tiempo real):

Con la llegada de Internet, llega Java y obtiene popularidad. A pesar de no ser adecuado, inicialmente, para la programación de tiempo real, se ha invertido recientemente mucho trabajo para producir versiones de Java para tiempo real (US National Institute of Standards and Technology, 1999; Bollella et al., 2000; J Consortium. 2000).

- Modula, Modula-2, PEARL, Mesa, CHILL, Java y Ada son todos ellos lenguajes de programación concurrente de alto nivel que incluyen aspectos orientados a ayudar al desarrollo de sistemas computacionales embebidos. El lenguaje occam, por contra, es un lenguaje mucho más reducido. No da un soporte auténtico para la programación de sistemas

grandes complejos, aunque proporciona las estructuras de control habituales, así como procedimientos no recursivos.

- El desarrollo de occam ha estado muy ligado al del *transputer* (May y Shepherd, 1984), un procesador con memoria integrada en el chip y controladores de enlaces que permiten construir fácilmente sistemas *multi-transputer.* Occam tuvo un importante papel en el campo emergente de las aplicaciones embebidas distribuidas débilmente acopladas.

Criterios generales de diseño de lenguajes

Aunque un lenguaje de tiempo real pudiera estar diseñado, en origen, para cumplir los requisitos de la programación de informática embebida, raramente se limita a esta área.
La mayoría de los lenguajes de tiempo real se emplean también como lenguajes para la implementación de sistemas para aplicaciones, tales como compiladores y sistemas operativos.
Young (1982) detalla los siguientes seis criterios, a vece discutidos, como la base del diseño de un lenguaje de tiempo real: seguridad, legibilidad, flexibilidad, sencillez, portabilidad y eficiencia. (En los requisitos originales de Ada aparece también una lista similar).

Seguridad

La seguridad del diseño de un lenguaje es la medida del grado en el que el compilador, o el sistema de soporte en tiempo de ejecución, pueden detectar automáticamente los errores de programación.
Obviamente, existe un límite en la cantidad y en los tipos de errores que pueden ser detectados por el sistema del lenguaje; por ejemplo, no es posible detectar de modo automático los errores de lógica del programador.
Un lenguaje seguro debe, en consecuencia, estar bien estructurado y ser legible, de modo que tales errores puedan ser localizados fácilmente.
Entre los beneficios de la seguridad encontramos:

- La detección temprana de errores en el desarrollo del programa, lo que supone una reducción general del coste.
- Las comprobaciones de tiempo de compilación no sobrecargan la ejecución del proceso, y un programa se ejecuta más veces de las que se compila.

El inconveniente de la seguridad es que puede llegar a complicar el lenguaje e incrementar tanto el tiempo de compilación como la complejidad del compilador.

Legibilidad

La legibilidad de un lenguaje depende de un conjunto de factores, como una elección adecuada de palabras clave, la facilidad para definir tipos, y los mecanismos de modularización de programas.
Según apunta Young (1982): «... el objetivo es ofrecer un lenguaje para la notación suficientemente claro, que permita que los principales conceptos del funcionamiento de un programa puedan ser asimilados tan sólo leyendo el texto del programa, sin tener que recurrir a diagrama de flujo y descripciones textuales auxiliares».
Los beneficios de una buena legibilidad son:

- Menores costes de documentación.
- Mayor seguridad.
- Mayor facilidad de mantenimiento.

El principal inconveniente es la mayor longitud de los programas.

Flexibilidad

Un lenguaje debe ser lo suficientemente flexible para permitir al programador expresar todas las operaciones oportunas de una forma directa coherente.
De otro modo, como con los lenguajes secuenciales anteriores, el programador tendrá que recurrir a funciones del sistema operativo o a fragmentos en código máquina para obtener el resultado deseado.

Simplicidad

La simplicidad es un objetivo nunca suficientemente alabado de cualquier sistema (ya sea la proyectada estación espacial internacional o una simple calculadora).
En los lenguajes de programación, la simplicidad aporta las ventajas de:

- Minimizar el esfuerzo de producción de compiladores.
- Disminuir la posibilidad de cometer errores de programación como consecuencia dc una mala interpretación de las características del lenguaje.

La flexibilidad y la simplicidad pueden ponerse también en relación con el **poder expresivo** (la posibilidad de ofrecer soluciones a un gran abanico de problemas) y con la **facilidad de uso** del lenguaje.

Portabilidad

Un programa deberá ser independiente, en la medida de lo posible, del hardware sobre el que se ejecuta.
Una de las principales autoproclamas de Java es que los programas se compilan una sola vez y se ejecutan en muy diversos sitios. Para un sistema de tiempo

real, esto es difícil de lograr -incluso con la llegada de los códigos binarios portables (Venners, 1999; XIOpen Company Ltd., 1996)-, dado que una parte importante de cualquier programa estará involucrada en la manipulación de recursos hardware.
A pesar de ello, un lenguaje deberá ser capaz de aislar la parte del programa que depende de la máquina de la parte independiente de la máquina.

Eficiencia

En un sistema de tiempo real, los tiempos de respuesta deberán estar garantizados; así pues, el lenguaje deberá permitir producir programas eficientes y predecibles.
Hay que evitar aquellos mecanismos que conduzcan a sobrecarga impredecibles del tiempo de ejecución.
Obviamente, los requisitos de eficiencia deberán estar compensados con los requisitos de seguridad, flexibilidad y legibilidad.

Prueba

Las pruebas deben ser extremadamente exigentes. Una estrategia de prueba completa incluirá muchas técnicas, la mayoría de las cuales son aplicables a cualquier producto de software.
El problema de los programas concurrentes de tiempo real es que los errores de sistema más difíciles de tratar surgen habitualmente de sutiles interacciones entre procesos.
A menudo, los errores dependen del instante de tiempo, y sólo se manifestarán en situaciones poco comunes.
Los métodos apropiados de diseño formal no evitan la necesidad de hacer pruebas: son estrategias complementarias.
Las pruebas, desde luego, no se restringen al sistema final ya ensamblado. La descomposición contemplada en el diseño y que se manifiesta en la modularidad del programa (y de los módulos) proporcionan una arquitectura natural para la prueba de componentes.
De particular importancia (y dificultad) en los sistemas de tiempo real es probar no sólo el correcto comportamiento en el entorno correcto, sino también comprobar la confiabilidad de su comportamiento en un entorno arbitrariamente incorrecto.
Habrá que contrastar todos los modos de recuperación e investigar los efectos de los errores simultáneos.
Como ayuda para cualquier tarea de prueba compleja, el disponer de un entorno de prueba presenta muchos atractivos. Para el software, tales entornos de prueba se denominan simuladores.

Simuladores

Un simulador es un programa que imita las acciones del sistema en el que se encuentra inmerso el software de tiempo real.
Simula la generación de interrupciones y efectúa otras acciones de E/S en tiempo real.
Empleando un simulador, podrán crearse situaciones de comportamiento normales y anormales. Incluso cuando el sistema final haya sido completado, determinados estados de error solo podrán ser experimentados de un modo seguro mediante un simulador.
Los simuladores pueden reproducir con exactitud la secuencia de eventos que se esperan del sistema real.
Es posible repetir experimentos en modos que normalmente son imposibles de obtener en el funcionamiento en vivo.
Desgraciadamente, para recrear de modo realista acciones simultáneas se hace necesario disponer de un simulado multiprocesador. Además, hay que tener en cuenta que para aplicaciones muy complicadas pudiera no ser posible construir un simulador apropiado.
Aunque los simuladores no exigen requisitos de muy alta fiabilidad, son, en cualquier aspecto, sistemas de tiempo real por propio derecho.
Ellos mismos deberán ser probados ampliamente, aunque eventualmente podamos tolerar algunos errores.
Pueden requerir hardware especial. (En el proyecto de la lanzadera espacial de la NASA los simuladores costaron más que el mismísimo software de tiempo real).
Finalmente, resulta un dinero bien gastado, por los muchos errores encontrados en el sistema durante las horas de «vuelo» simulado.

Prototipado

La aproximación estándar en «cascada» al desarrollo del software (esto es, requisitos. especificación, diseño, implementación, integración, prueba y despliegue o implantación) con lleva el problema fundamental de que los errores en los requisitos iniciales o en las fases de especificación sólo se reconocen al entregar el producto (o en el mejor de los casos, durante las pruebas).
Corregir estas carencias en esta etapa tan tardía es muy largo y costoso.
El prototipado intenta descubrir estos fallos lo antes posible cuando presentamos al cliente una portada del sistema.
El principal objetivo de un prototipo es ayudar a cerciorarnos de que en la especificación de requisitos ha sido captado lo que realmente quiere el cliente.
Esto tiene dos aspectos:

- ¿Es correcta la especificación de requisitos (en términos de lo que desea el cliente)?

- ¿Es completa la especificación de requisitos (ha incluido el cliente todo lo que desea)?

Uno de los beneficios de la puesta en marcha de un prototipo es que el cliente puede experimentar situaciones que han sido entendidas sólo vagamente.
Es casi inevitable que en el desarrollo de esta actividad haya que realizar cambios en los requisitos.
Si las técnicas de especificación empleadas no son formales, el prototipado puede utilizarse para darnos confianza en la consistencia de nuestro diseño global.
Así, si usamos un gran diagrama de flujo, un prototipo podría ayudarnos a comprobar que todas las conexiones necesarias están en su lugar y que todos los fragmentos de datos que fluyen por el sistema visitan efectivamente las actividades requeridas.
La desventaja más evidente de estos prototipos es que no tienen en cuenta los aspectos de tiempo real de la aplicación. Para obtener esto, necesitamos una simulación del sistema.

Interacción hombre – máquina

Todos los sistemas de tiempo real incluyen o afectan a los seres humanos. La mayoría, de hecho, involucran comunicación directa entre el software en ejecución y uno o más operadores humanos.
Como el comportamiento humano introduce la mayor de las fuentes de variabilidad en un sistema en funcionamiento, es lógico que el diseño del componente HCI *(Human-Computer Interaction;* interacción hombre-máquina) sea uno de los más críticos de toda la estructura.
La línea de investigación sobre los principios de diseño en los que basar la construcción de componentes HCI es muy activa.
Tras haber estado en segundo plano durante muchos años, el HCI es apreciado hoy en día como un punto central para la producción de software bien diseñado (de cualquier software).
La primera cuestión importante es la modularidad; las actividades de HCI deben encontrarse aisladas y con especificaciones de interfaces bien definidas.
Estas interfaces constan a su vez de dos componentes, siendo las funciones de cada una de ellas bien diferentes.
En primer lugar, se definen aquellos objetos que se transfieren entre el operador y el software.
En segundo lugar (muy importante), es necesario especificar cómo hay que presentar y recibir estos objetos dei usuario.
Una definición rigurosa del primer tipo de componente de la interfaz debe incluir predicados sobre las acciones permitidas para un operador en cada estado del sistema. (Ej: ciertas operaciones podrían requerir autorización (o sólo podrían ser llevadas a cabo sensatamente o de modo seguro) cuando el sistema estuviera

correctamente dispuesto. Por poner un ejemplo, un avión a control remoto ignorará una operación que le conduzca fuera de su ruta de vuelo segura.

Una importante cuestión de diseño en cualquier sistema interactivo es quién está al mando.

En un extremo, la persona podría dirigir explícitamente al computador para que realizara funciones concretas, mientras que en el otro extremo el computador podría tener el control completo (aunque eventualmente podría solicitar información al operador humano).

Obviamente, la mayoría de los sistemas de tiempo real estarán entre estos dos extremos; se les conoce como sistemas de **iniciativa mixta.**

En algunas ocasiones el usuario está al mando (por ejemplo, cuando se proporciona un nuevo mandato), y en otras el sistema controla el diálogo (por ejemplo, cuando se extraen los datos necesarios para efectuar una operación que se acaba de solicitar).

El principal objetivo en el diseño de la componente de interfaz es la captura de todos los errores derivados del usuario en el software de control de la interfaz, y no en el software de aplicación o en el resto del sistema de tiempo real. Si esto es así, el software de aplicación podrá presuponer un **usuario perfecto,** y en consecuencia su diseño e implementación se harán más simples (Burns, 1983).

En nuestra discusión, el término **error** se refiere a una acción no intencionada. Reason (1979) lo denomina **lapsus** (slip). y usa el término **equivocación** para referirse a una acción errónea deliberada.

Aunque una interfaz pueda ser capaz de bloquear las equivocaciones conducentes a situaciones peligrosas, no será posible reconocer como una equivocación un cambio serio de un parámetro de la operación.

La única forma de eliminar estos errores es mediante la formación apropiada y la supervisión de los operadores.

Aunque estos lapsus ocurrirán inevitablemente (y en consecuencia debemos protegemos contra ellos), su frecuencia puede reducirse de modo significativo si el segundo componente de la interfaz, el auténtico módulo de entrada/salida del operador, está bien definido.

En este contexto, sin embargo, el término «bien definido» no es fácil de establecer.

La especificación de un módulo de E/S cae esencialmente bajo la competencia de la psicología.

El punto de arranque es un modelo del operador o usuario final.

De la comprensión de estos modelos, podemos establecer ciertos principios de diseño, entre los que encontramos estos tres importantes:

- Predecibilidad: es preciso proporcionar datos suficientes al usuario para que éste conozca de modo exacto los efectos de cada operación a partir del conocimiento sobre esta operación.
- Conmutatividad: el orden en que el usuario cumplimenta los parámetros de una operación no deberá afectar al sentido de la operación.

- Sensibilidad: si el sistema puede presentar diferentes modos de operación, deberá mostrarse siempre el modo actual.

Los sistemas de tiempo real tienen la dificultad añadida de que los operadores humanos no son los únicos actores que cambian el estado del sistema.
Una interrupción puede causar un cambio de modo, y la naturaleza asíncrona de la entrada de operación podría ocasionar que un mandato, que era válido cuando el operador lo inició, no esté disponible para el nuevo modo.
Gran parte del trabajo sobre HCI ha estado relacionado con el diseño de pantallas para la presentación no ambigua de datos y para que la captura de datos de entrada se efectúe con el menor número de pulsaciones.
Existen, sin embargo, otros factores relacionados con la ergonomía de la estación de trabajo, así como cuestiones más amplias sobre el entorno de trabajo de los operadores.
Los diseños de pantallas deben ser «probados» mediante un prototipo, pero la satisfacción en el trabajo es una métrica que solamente podemos aplicar sobre intervalos de tiempo de mayor duración que los que se ofrecen al experimentar con un prototipo.
Puede que los operadores trabajen en turnos de ocho horas, cinco días a la semana, año tras año.
Si además añadimos un trabajo agobiante (como el de controlar el trafico aéreo), la satisfacción en el trabajo y las prestaciones se convierten en factores críticos que dependen de:

- La multitud de tareas interdependientes que hay que llevar a cabo.
- El nivel de control de que dispone el operador.
- El grado en que el operador comprende la operación del sistema completo.
- El número de tareas constructivas que se le permite realizar al operador.

La comprensión puede mejorarse si presentamos siempre al operador una imagen de «qué está pasando».
Y ciertamente, es posible construir sistemas de control de procesos que ilustren el comportamiento del sistema en forma gráfica, pictográfica, o como una hoja de cálculo.
El operador puede experimentar con los datos para modelar formas de mejorar las prestaciones del sistema.
Es muy razonable incorporar tales **sistemas de ayuda a la decisión** en el bucle de control, de modo que el operador pueda ver el efecto de cambios menores sobre los parámetros de la operación.
De esta forma, se puede incrementar la productividad y mejorar el entorno de trabajo de los operadores.
Se reduce el número de equivocaciones que se cometen.

Gestión del Diseño

Sólo obtendremos un producto fiable y correcto si las actividades de especificación, diseño, implementación y prueba se llevan a cabo con una completa y alta calidad.
Hay un amplio rango de técnicas que pueden ayudar a lograr esta calidad.
El uso apropiado de un lenguaje de alto nivel bien definido.
Gestionar adecuadamente tanto la programación como el diseño. Los lenguajes de programación concurrente modernos tienen un papel importante en este proceso de gestión.
La clave para lograr la calidad descansa en la adecuada verificación y validación. En cada etapa de diseño e implementación, es preciso usar procedimientos bien definidos para asegurar que las técnicas necesarias se realizan correctamente.
Donde se precisen fuertes garantías, pueden usarse demostradores de teoremas y comprobadores de modelos para verificar formalmente los componentes especificados. Más ampliamente se utilizan muchas otras actividades estructuradas; (Ej.: análisis de peligros, análisis de árboles de fallos en software, análisis de modos de fallo y de efectos, e inspecciones de código, etc.).
Cada vez hay más necesidad de herramientas de software como ayuda para la validación y para la verificación. Tales herramientas realizan un trabajo importante, y su uso puede reducir la necesidad de inspección humana. Así pues, estas herramientas deben ser de una calidad extremadamente alta.
Las herramientas de apoyo, sin embargo, no pueden substituir a un equipo de personas experimentado y bien entrenado.
La aplicación de técnicas rigurosas de ingeniería de software y el empleo de las características del lenguaje establecen la diferencia en cuanto a sistemas de tiempo real de calidad y asequibles.
Muchas muertes han sido atribuidas a errores de software. Es posible detener una futura epidemia, pero sólo si la industria se aparta de los procedimientos ad ***hoc,*** los métodos informales, y los lenguajes inapropiados de bajo nivel.

Bibliografía

- Plantillas de Clase. Mgter Vallejos, Oscar A.
- Ian Sommerville. Ingeniería del Software. 7ma.Edición. Pearson Educación. 2005.
- Roger S. Pressman Ingeniería del Software. 7ma. Edición. McGraw Hill. 2010.
- Alan BURNS y Andy WELLINGS. Sistemas de Tiempo Real y Lenguajes de Programación. 3ª Edición. Editorial: ADDISON-WESLEY. ISBN: 8478290583. 2005.

- David Vallejos Fernández, y otros. Programación Concurrente y Tiempo Real [3ª Edición]. 2015. Createspace Independent Publishing Platform. ISBN: 978-1518608261.