Tema 2 Lenguajes para aplicaciones en tiempo real

2.1 Introducción

Una vez vistas las características principales de los sistemas en tiempo real (STR), ya podemos enumerar las características generales del software de tiempo real

1. Debe ser extremadamente fiable. Un fallo del sistema puede ser muy caro en términos de pérdida de producción (caso de un control de proceso), o en términos de la propia vida humana (navegación aérea, control de una central de energía nuclear).
2. Deberá ser capaz de manejar procesos que evolucionan en paralelo. Exceptuando los sistemas más sencillos lo habitual será dividir la aplicación en *tareas* o *procesos* que pueden trabajar de forma cooperativa o incluso competir por los recursos.
3. Es generalmente voluminoso y complejo. Esto, combinado con el problema de la fiabilidad, significa que los costes de mantenimiento y desarrollo pueden ser muy elevados.
4. Responderá a una variedad de eventos externos con tiempos de respuesta garantizados (comportamiento determinista en la medida de lo posible).
5. Admitirá la conexión con una gran variedad de dispositivos de E/S no estándares, así como con los periféricos convencionales de un ordenador.
6. Debe ser eficiente para poder hacer la mejor utilización del equipo *hardware* disponible.

La producción y el mantenimiento de tal software supone fijar unos fuertes requisitos sobre los lenguajes de programación. Los primeros STR fueron programados usando lenguajes ensambladores. Este tipo de codificación aun es usado para sistemas pequeños que requieren muy alta velocidad de cómputo. Los problemas intrínsecos a la programación en ensamblador hacen muy incómodo y difícil el desarrollo de sistemas en tiempo real. Posteriormente se aplicaron lenguajes de alto nivel, digamos convencionales, que no dieron los resultados apetecibles. Estos hechos propiciaron el desarrollo de nuevos lenguajes para la programación de STR.

Si el lenguaje utilizado no incorpora estas cualidades necesitaremos del apoyo del sistema operativo para poder desarrollar nuestras aplicaciones en tiempo real.

2.2 Características de los lenguajes de tiempo real

Las características que un usuario busca en un lenguaje en tiempo real se pueden dividir en áreas generales:

1. Seguridad
2. Capacidad de manejar el tiempo
3. Expresión de la concurrencia
4. Legibilidad
5. Flexibilidad
6. Simplicidad
7. Portabilidad
8. Eficiente

2.2.1 Seguridad

La seguridad de un lenguaje es la medida de hasta qué punto los errores de programación pueden ser detectados automáticamente por el compilador o por el sistema soporte del programa ejecutable. En general, el diseño de un lenguaje debería ser tal que se detectaran tantos errores como fuera posible en tiempo de compilación, en lugar de en tiempo de ejecución. Esto es deseable por dos razones: la primera es que cuanto antes se detecten los

errores en el proceso de desarrollo, menores serán los costes generales. La compilación puede efectuarse sobre cualquier máquina que soporte el lenguaje, mientras que las pruebas en tiempo de ejecución deben realizarse sobre el hardware o equipo real (o una simulación de éste). En segundo lugar, la prevención de errores en tiempo de ejecución requiere a menudo la inserción de código adicional en el programa (normalmente por el compilador), lo que conduce a sobrecargas adicionales en términos de velocidad de ejecución y utilización de memoria.

Algunas de las características que mejoran la seguridad de un lenguaje son:

- Estructura de los programas bien modulada
- Fuerte tipado de las variables
- Inclusión de rangos en los tipos y subtipos de datos
- Sintaxis que no dé lugar a ambigüedades

Por otra parte es conveniente que un lenguaje permita el manejo de excepciones para que, si se detectan errores en tiempo de ejecución, se puedan llevar a cabo medidas correctoras.

Obviamente, no se pueden detectar automáticamente los errores en el diseño lógico de un programa. Sin embargo, existen menos probabilidades de que ocurran si el lenguaje estimula al programador para que escriba unos algoritmos claros y bien estructurados, facilitando unas construcciones adecuadas.

2.2.2 Capacidad para manejar el tiempo

Dentro de las tareas que puede ser preciso ejecutar habrá un número de ellas de naturaleza periódica, que solo se ejecutarán cada cierto tiempo. Mientras no sea preciso ejecutarlas deberán bloquearse para no hacer gasto de CPU. Por otra parte otras tareas se ejecutarán a instantes del día determinados, por lo que será preciso que tengan acceso al reloj con la hora del día.

Hay lenguajes que para poder expresar el tiempo precisan de llamadas al sistema operativo. Esto, además de ralentizar el sistema, puede conllevar tener relojes de poca resolución. Es conveniente por tanto tener accesos a los relojes hardware para poder hacer temporalizaciones precisas.

Por último puede ser muy útil saber cuál es el tiempo límite de una tarea y cuánto le falta para alcanzarlo. Este tipo de información puede resultar muy útil a la hora de decidir la tarea que se planificará. Esta última faceta en el manejo del tiempo es la más difícil de conseguir y muy pocos lenguajes o sistemas operativos la implementan

2.2.3 Expresión de la concurrencia

Dado que los sistemas en tiempo real caen dentro de la gama de los sistemas reactivos será preciso responder a los cambios que se producen en el entorno. Este entorno puede ser complejo y con múltiples elementos, que evolucionan simultáneamente, por lo que resulta conveniente hacer una división del control del sistema en tareas.

Típicamente la concurrencia se expresa a través del uso de los sistemas operativos, pero también se puede realizar por medio del lenguaje de programación. El desarrollo de la programación concurrente va mucho más allá de la mera creación de tareas; resulta fundamental poder coordinar y sincronizar estas tareas mediante los recursos apropiados. Otro problema es el de la compartición de la información de forma segura, la forma en que se otorgan recursos, la política de planificación que se va a seguir, si se pueden detectar de forma automática los estancamientos o los estados de inanición, etc.

Diferentes lenguajes o sistemas operativos ofrecen variadas alternativas al menos a algunos de estos problemas.

2.2.4 Legibilidad

La legibilidad de un lenguaje depende de gran variedad de factores, que van desde la elección de las palabras clave (o reservadas) hasta las facilidades para la modularización de los programas. La legibilidad es una medida de la facilidad con que pueden entenderse las operaciones de un programa sin necesidad de recurrir a descripciones escritas u organigramas.

Los beneficios de una buena legibilidad son:

- Los costes de documentación se reducen si la pieza central de documentación es el propio programa.
- Se facilita la comprensión de un programa, y, por tanto, los errores pueden ser localizados más fácilmente. Como consecuencia, aumenta la fiabilidad.
- Se facilita el mantenimiento del programa. Cuando hay que modificar un programa, es frecuente que la persona encargada de la modificación no sea la misma que lo desarrolló originalmente.

El logro de una buena legibilidad requiere la cooperación del programador: es posible escribir programas ilegibles en cualquier lenguaje.

El diseño del lenguaje utilizado juega también un importante papel. En los niveles inferiores de la construcción del programa, el lenguaje debe aportar facilidades para poder especificar qué datos están implicados en el cálculo y cómo deben utilizarse. Es decir, debe suministrar tipos de datos adecuados y facilitar la elección de identificadores adecuados. Las estructuras de control han de ser fácilmente comprensibles. En los niveles superiores de la construcción del programa, el lenguaje debe aportar las posibilidades de modularización, de modo que se pueda componer un programa a partir de cierto número de unidades operativas con entidad propia.

2.2.5 Flexibilidad

Un lenguaje debe proporcionar al programador la suficiente flexibilidad para que pueda expresar la mayoría de las operaciones requeridas sin tener que recurrir a añadidos en código máquina, o a construcciones muy complicadas. Un campo donde se precisa de gran flexibilidad es la programación de dispositivo de E/S. Dotar a los lenguajes de flexibilidad total puede entrar en conflicto con la seguridad, por ejemplo permitir que un puntero señale a distintos tipo de datos. Habitualmente los aspectos de E/S se encapsulan en módulos aparte para que puedan ser bien probados y revisados.

2.2.6 Sencillez

La sencillez de un lenguaje reduce los costes asociados al adiestramiento del programador, así como las posibilidades de cometer errores de programación como resultado de una errónea interpretación de las características del lenguaje. Asociada con la sencillez está la consistencia: un buen lenguaje no debería imponer restricciones arbitrarias en el uso de sus características.

2.2.7 Portabilidad

Un diseño de lenguaje que sea independiente del hardware permitirá que el software pueda ser trasladado de una máquina a otra con relativa facilidad. La portabilidad es más difícil para STR, puesto que hacen uso habitualmente de características específicas del hardware. Una solución práctica consiste en aceptar que un programa para un STR no será directamente transportable, y en restringir las áreas no transportables (que incluirán código en bajo nivel) a módulos específicos.

2.2.8 Eficiencia

Los STR deben procurar frecuentemente un alto grado de flujo interno de cálculo para satisfacer las restricciones impuestas por el equipo externo a monitorizar o a controlar. Un diseño de lenguaje debe, por tanto, ser capaz de una implantación eficiente.

En el diseño de los primeros lenguajes en tiempo real, la eficiencia se consideraba de capital importancia, con el resultado de que, para conseguirla, se sacrificaba la seguridad, flexibilidad y sencillez. La caída en los costes del hardware y el incremento en la velocidad de cómputo de los ordenadores han cambiado el orden de prioridades. La motivación principal en el diseño de un lenguaje moderno es facilitar la escritura de un software fiable. De la eficiencia se encargarán principalmente los compiladores y sistemas operativos.

2.3 Requisitos de los lenguajes para los STR

Para que un lenguaje pueda estar dotado de las características mencionadas en el apartado anterior, debe cumplir una serie de requisitos. En este apartado avanzaremos una serie de ellos, que serán desarrollados con mayor profundidad posteriormente, tomando el lenguaje Ada como base.

2.3.1 Declaración de variables y constantes

El propósito de declarar una entidad usada en un programa es dar al compilador información para reservar el espacio adecuado de memoria e informar explícitamente al sistema de los nombres usados. Lenguajes como Pascal, Modula-2 y Ada requieren que todos los objetos sean declarados explícitamente y que se les asocie un tipo de datos en la declaración. La provisión de un tipo permite al compilador chequear que el objeto sea utilizado sólo en operaciones asociadas a ese tipo.

Algunos lenguajes más antiguos, como BASIC y FORTRAN, no exigen declaraciones explícitas. En FORTRAN, si los objetos no son declarados, se les asocia un tipo implícitamente: por ejemplo, nombres que empiecen por I-N corresponden a variables enteras; el resto, a variables reales. Esta declaración implícita puede dar lugar a la construcción de programas sintácticamente correctos pero funcionalmente erróneos. Considerar el siguiente fragmento de programa:

```
100 ERROR = 0
    ...
200 IF X=Y THEN GOTO 300
250 EROR=1
300 ...
400 IF ERROR=0 THEN GOTO 1000
```

En FORTRAN, o BASIC, ERROR y EROR serán consideradas dos variables diferentes, mientras que la intención del programador fue utilizar sólo la variable ERROR, y cometió un error al teclear.

La declaración implícita también puede dar lugar a confusión y malinterpretación. Considerar el siguiente fragmento:

```
REAL KP, KD, KI
INTEGER DACV, ADCV
...
100 DACV = KP*(KD/KI)
```

Si el lector se salta las declaraciones iniciales, al leer la línea 100 puede asumir que las variables de la derecha son enteras y que el resultado es convertido a real, y almacenado en una variable real, cuando en realidad el programa está haciendo justamente lo opuesto.

El ejemplo más notorio de la falta de seguridad en los programas escritos en FORTRAN es el error que causó un mal direccionamiento en la nave *Voyager*. La sentencia que se pretendía escribir era:

```
DO 20 I=1,100
```

que es una construcción de un bucle.

La sentencia que se escribió fue:

```
DO 20 I=1.100
```

Debido a que los espacios no son tenidos en cuenta, el compilador trató a los caracteres a la izquierda del signo = como un nombre de variable real (DO20I) y le asignó el valor 1.1.

También es conveniente que se den valores iniciales a las variables, o que al menos el compilador verifique antes de usar una variable que ésta ha sido previamente inicializada.

2.3.2 Tipos de datos

El tipo de datos de un objeto especifica el rango de valores que puede asumir el objeto y el conjunto de operaciones que se le pueden aplicar. La riqueza de tipos aportada por un lenguaje y el grado de rigor aplicado a la compatibilidad de tipos mejoran la seguridad y claridad de un programa. Un lenguaje con tipología fuerte no

permite asignar un valor de un tipo a un objeto de otro tipo, incluso cuando las representaciones que subyacen en los dos tipos sean idénticas.

Un tipo debería especificar una clase de objetos lógicamente relacionados que serían disjuntos de todas las demás clases de objetos, aunque compartan un tipo subyacente común. Por ejemplo, un programa puede contener una variable entera que represente un índice de array y otra variable entera que represente un número de años. Tales objetos deberían ser de tipos distintos, de forma que se pudieran prevenir operaciones ilegales, tales como la asignación de un valor índice a un objeto que representa los años de alguien. Esto se logra si el lenguaje permite la definición de nuevos tipos derivados de los ya existentes. Se podrían definir los tipos *edad* e *índice*, en términos del tipo *integer* predefinido. El compilador detectaría en este caso el error de asignación mencionado. La posibilidad de definir tipos enumerados aumenta también la claridad y seguridad de los programas.

La seguridad se mejora aún más si el rango de valores permitido para cada objeto de datos se restringe estrictamente a los necesarios (tipos subrango). Por ejemplo, si se utiliza una variable de tipo entero para representar segundos, su rango puede especificarse para que valga de 0 a 5 El compilador podría detectar una asignación estática incorrecta, fuera de ese rango.

Una tipología fuerte ofrece una mayor seguridad a cambio de una pérdida de flexibilidad.

2.3.3 Estructura de programas

Las facilidades ofrecidas normalmente por un lenguaje para la estructuración de programas pueden dividirse en dos niveles. En el primero, se necesitan estructuras de control para especificar las acciones básicas de un programa. En el segundo, se precisan estructuras para agrupar conjuntos de acciones lógicamente relacionadas en unidades simples. Ambas facilidades constituyen las herramientas básicas de la programación estructurada.

En cuanto al primer nivel, un aspecto contencioso es la facilitación de una sentencia *exit* para permitir la terminación anormal de un bucle. Con frecuencia, conduce a obtener estructuras más simples y eficientes. No obstante, viola el principio de "una sola entrada/una sola salida" de la programación estructurada, por lo que su inclusión en el lenguaje es discutible.

En relación con el segundo nivel, el lenguaje debería proporcionar mecanismos de abstracción, con los que obtener mayor seguridad y portabilidad; es decir, debería facilitar una construcción para englobar datos y las acciones a efectuar sobre los datos en una unidad, con una interfaz de comunicación.

2.3.3.1 Modularidad y variables:

Al tener el programa dividido en módulos establecemos compartimentaciones, podemos hablar de:

- Alcance y visibilidad
- Variables globales y locales
- Control de la visibilidad y el alcance.

El alcance de una variable es la región del programa donde la variable es potencialmente accesible y modificable. La región en la que una variable puede realmente ser modificada es su visibilidad. Muchos lenguajes emplean el bloque como unidad del alcance de una variable.

Es también habitual que los procedimientos puedan crear variables locales, existentes solamente mientras se ejecuta el subprograma. También suele ser frecuente poder contar con variables que se puedan compartir por distintos procedimientos. Esto no resulta tan común si los procedimientos pertenecen a distintos procesos, en este caso la compartición puede resultar más compleja.

Los lenguajes modernos ofrecen características que permiten controlar la visibilidad y el alcance de una variable modificándolo en caso de ser preciso.

2.3.4 Multiprogramación

Para describir y construir un sistema software en función de una serie de tareas concurrentes se necesita disponer de facilidades de multiprogramación. Estas facilidades deben permitir especificar procesos software y proporcionar algún medio de comunicación y sincronización entre ellos.

Una solución para lograr comunicación y sincronización entre procesos, sin que el lenguaje tenga que aportar ninguna construcción específica para la multiprogramación, consiste en que todas las facilidades necesitadas vengan soportadas por un sistema operativo multitarea. El programa accederá a esas facilidades a través de un conjunto de llamadas al sistema.

Esta solución, atractiva por su sencillez, tiene una serie de inconvenientes: la comunicación y sincronización son inseguras y de un nivel demasiado bajo; es fácil hacer una mala utilización de las funciones del S.O. La operación de los programas que las utilizan puede ser de difícil comprensión. Los sistemas construidos de este modo son raramente transportables, ya que dependen del S.O. utilizado, y éste dependerá de la máquina.

Otra solución a la multiprogramación consiste en introducir dentro del propio lenguaje construcciones dedicadas para la especificación de procesos y para la comunicación y sincronización entre procesos.

2.3.5 Programación de dispositivos de bajo nivel

Una de las características principales de los STR es que incorporan una serie de dispositivos de E/S de propósito especial. Los lenguajes de tiempo real deberían incluir facilidades alto nivel para programación de estas funciones de bajo nivel, entre otras:

- Facilidad para acceder a los bits individuales de los registros de E/S.
- Facilidad para encapsular el código no transportable en unidades claramente identificables.
- Un dispositivo de E/S puede ser considerado como un procesador que realiza una tarea fija. El driver del dispositivo puede ser tratado como un proceso software, uno de los varios procesos concurrentes que modelizan el sistema.

2.3.6 Tratamiento de errores

Las especificaciones de alta fiabilidad de un STR implican que un lenguaje de tiempo real debe facilitar mecanismos que permitan responder a condiciones de error y programar acciones de recuperación.

Una solución para el tratamiento estructurado de errores se basa en las ideas de excepción y sus manejadores. Una excepción es una especie de indicador de error que está *on* u *off*. Si está *on*, indica la ocurrencia de una condición de error. La ejecución del subprograma en curso finaliza y el control se transfiere al subprograma manejador del error.

2.4 Introducción al lenguaje Ada

Ada es un lenguaje de alto nivel, originalmente patrocinado por el Dpto. de Defensa de EEUU. para ser utilizado en los llamados sistemas incrustados.

A principios de los años 70, el Dpto. de Defensa, tras un estudio de las causas de sus altos gastos en software, descubrió que más del 50% de los costes se debían a los sistemas incrustados, y a la falta de estándares en el desarrollo de los mismos. Se gastaban grandes cantidades de dinero en compiladores, por el elevado número de lenguajes que se utilizaban, en aprendizaje y mantenimiento asociados a esa falta de estandarización.

El primer paso hacia el desarrollo de un estándar fue la escritura de un documento que recogía los requisitos a cumplir. La primera versión se llamó STRAWMAN y fue publicada en 1975. Se refinó, tras recibir comentarios de varias fuentes, y pasó a llamarse WOODENMAN. Una iteración posterior produjo TINMAN, en 1976 En este punto, muchos lenguajes existentes se evaluaron frente a TINMAN. Esta evaluación clasificó a los lenguajes en tres categorías:

- NO APROPIADOS: FORTRAN, CORAL 66,... Desechados.
- NO INAPROPIADOS: RTL/2, LIS,... Incluían algunas características interesantes.
- BASE RECOMENDADA: PASCAL, PL/I, ALGOL 6 Puntos de partida para diseñar el lenguaje final.

El documento fue revisado y reorganizado, dando lugar a IRONMAN. Se elaboraron proyectos para diseñar un nuevo lenguaje, a partir de las bases recomendadas. De estos proyectos, fueron elegidos cuatro para ser

desarrollados en paralelo y en competición. A partir del juicio de muchos grupos de todo el mundo, fueron elegidos dos proyectos, a los que se dio un año de plazo para refinar sus diseños. También se refinaron los requisitos, dando lugar a STEELMAN, el documento final.

La elección final fue hecha el 2 de mayo de 1979 El proyecto ganador fue el lenguaje desarrollado por CII Honeywell Bull. Se le dio el nombre de Ada, en honor de lady Ada Augusta Byron, condesa de Lovelace (1815-52), hija de lord Byron, que fue la ayudante de Charles Babbage. Ella fue la primera programadora.

La siguiente fase en el desarrollo del Ada consistió en la evaluación del lenguaje llevada a cabo por varios equipos de EEUU y Europa. Después de ser revisado, en julio de 1980 se publicó la primera versión del Ada. En 1983, el manual de referencia del lenguaje Ada fue aprobado como un estándar ANSI (American National Standards Institute). El efecto de esta estandarización fue promover el Ada fuera del dominio exclusivo de Defensa, hacia la industria computacional general. En el verano de 1985, los países de la NATO (North Atlantic Treaty Organization) aprobaron un convenio que obligaba al uso del Ada para el desarrollo de todos sus sistemas. En 1987 Ada pasó a ser un estándar ISO (International Standard Organization).

2.4.1 Estructura de un programa en Ada

Un programa en Ada está compuesto de una o más unidades, la mayoría de las cuales pueden ser compiladas por separado. Una unidad consta de subprogramas, tareas (tasks), paquetes (*packages*) y/o unidades genéricas.

Un subprograma es un *procedure* o una función, y expresa una acción secuencial. Una tarea define una acción que puede ser ejecutada en paralelo junto con otras tareas. Un grupo de tareas puede ser implementado en un solo procesador, en un multiprocesador, o en un sistema distribuido.

Un paquete (*package*) es una colección de recursos computacionales relacionados. Agrupa definiciones de tipos de datos, variables, subprogramas, tareas u otros packages. Su función básica es ayudar al programador a conseguir abstracciones lógicas. Consta de dos partes: especificaciones y cuerpo. La parte de especificación identifica la información visible al cliente, es decir, es su interfaz. El cuerpo contiene los detalles de implementación, que deben ser ocultados al cliente. Ambas partes pueden ser compiladas por separado.

Una unidad genérica sirve como mecanismo para construir componentes software reutilizables (utilizables por objetos de diferentes tipos).

BIBLIOGRAFIA

En la elaboración de este tema se han consultado los siguientes libros:

[1] S.J. Young; Lenguajes en Tiempo Real; Ed. Paraninfo 8 Caps. 1,9

[2] J.G.P. Barnes; Programming in Ada; Ed. Addison-Wesley 8 Caps. 1,2

[3] G. Booch; Software Engineering with Ada; The Benjamin/Cummings Publishing Company 8 Caps 1-3

[4] S.Bennet; Real Time Computer Control; Ed. Prentice-Hall 9 Cap. 5.