

ADA COMO LENGUAJE
DE ESPECIFICACION

Juan Grompone

Montevideo - Uruguay

1. INTRODUCCION

Este trabajo se basa en la experiencia recogida en la especificacion, y posterior implementacion, de una Central Telex concebida como un conjunto distribuido de procesadores. Este proyecto, realizado durante 1985, era la tercera experiencia de diseño de un equipo de este uso, de modo que el tema era conocido y tambien eran conocidas las dificultades de ingenieria de Software que se encontraban en un proyecto de esta naturaleza.

El diseño original del Sistema era de 1978-1980 [1]. Debido a la falta de experiencia en proyectos de esta dificultad, la metodologia de trabajo fue mala. En 1981-1982 [2] se realizo un rediseño importante y se mejoraron muchos aspectos de la ingenieria de Software. En ambas oportunidades se habian empleado tecnicas convencionales para el diseño y la documentacion. Las definiciones del proyecto eran informales y se reunian en "carpetas de diseño" que contenian anotaciones manuscritas, diagramas con "rombos y rectangulos", esquemas, fragmentos de programas y toda suerte de material no homogeneo.

La segunda experiencia mostro que eran necesarios metodos mas formales de trabajo. Las principales ideas de rediseño fueron publicados en [3]. De aqui surgio la conviccion que los nuevos proyectos emplearian una base teorica mas importante.

En la presente experiencia se decidio desde un principio elaborar un documento formal, conocido como Manual de Especificacion, en el que se establecieron reglas precisas. Estas reglas se pueden resumir en la frase: toda la especificacion se realizaria con programas ADA [4,5].

El Manual de Especificacion llego a la version 11 a lo largo de sus sucesivos refinamientos. En los cuatro primeros capitulos se especificaba el Hardware del Sistema, con especial enfasis en el diseño de la programacion. En los cinco siguientes se especificaba la programacion de Tiempo Real. En los tres ultimos se especificaba el Lenguaje Hombre Maquina que usaban las Consolas del Sistema.

En este trabajo se analizan las tecnicas y los resultados de esta experiencia, considerada altamente positiva en todos sus aspectos.

2. VENTAJAS DE ESPECIFICAR EN UN LENGUAJE CON COMPILADOR

Si bien muchas veces se ha sostenido la importancia de emplear un lenguaje formal de especificacion para los problemas de computacion [7], en este trabajo deseamos poner enfasis en el empleo de un lenguaje de especificacion compilable, de uso general, frente a especificaciones verbales, diagramas o lenguajes creados por el diseñador.

En forma breve, una especificacion formal, realizada en un lenguaje de alto nivel (es decir, alto nivel respecto al nivel de implementacion) permite abstraer los elementos esenciales y separar así diseño de implementacion. Es un punto esencial del problema la eleccion del lenguaje de especificacion de manera de lograr que esta separacion sea efectiva.

Un lenguaje formal que posee un compilador asegura la coherencia y la completitud de la especificacion: al compilar la especificacion se detectan todos los errores formales. Este tipo de verificacion no puede ser realizada con un lenguaje creado para una aplicacion y, la mayoría de las veces, poco formal (en el caso de ADA se disponia solamente de un compilador que implementaba parcialmente el lenguaje, pero de todos modos presto gran utilidad en el proyecto [6]).

Con caracter general, las especificaciones formales permiten, además, algunas de las siguientes ventajas:

- definir en forma precisa algoritmos, estructuras de datos, nombres simbolicos y las acciones sobre estos objetos.

- reemplazar diagramas o descripciones verbales por objetos mas precisos.

- reemplazar las descripciones funcionales -por ejemplo del Hardware- por descripciones mas cercanas al implementador.

Estas características justifican el empleo de lenguajes formales en general, pero la propuesta de ADA requiere argumentos adicionales. Las razones para elegir ADA como lenguaje de especificacion se pueden resumir en los siguientes puntos:

- el objetivo principal del lenguaje es la confiabilidad y no la facilidad de uso: esta mas cerca de un lenguaje de especificacion que de implementacion.

- es un lenguaje "grande", con muchas primitivas y opciones lo cual da una gran flexibilidad para un

especificador. Vale la pena notar que esta característica es frecuentemente señalada como un inconveniente de ADA porque se lo contempla desde el punto de vista del fabricante de compiladores o del hardware a utilizar [8,9].

- es un lenguaje fuertemente orientado a tipos, a la organización modular y es muy exigente en sus aspectos formales.

- deriva de PASCAL y de allí que no se requiera un entrenamiento especial para comprender la especificación.

- posee primitivas de tiempo real. Con el desarrollo de compiladores y aumento de potencia de los microprocesadores se convertirá en un lenguaje de mucha difusión en esta área.

- puede suponerse que posee mucho futuro como lenguaje -mas que lenguajes ocasionales definidos por implementaciones- puesto que es un lenguaje moderno, impulsado por el Departamento de Defensa de los Estados Unidos de Norte America.

En lo que sigue de este trabajo se presentan ejemplos concretos que ilustran estas afirmaciones.

3. EJEMPLOS DE ESPECIFICACION DE DATOS

Si bien se suele trabajar con descripciones bastante precisas de las estructuras de datos, muchos lenguajes de implementación permiten trabajar libremente con los tipos, ya sea porque no imponen restricciones como ocurre con el Assembler, ya sea porque no hay estructuras de tipos abiertas y se deben emplear solamente enteros, caracteres, etc. Esta es una de las fuentes de errores más importantes en el diseño y la implementación de sistemas [10].

En el ejemplo que consideramos se manejan ciertos tipos de octetos, los caracteres Baudot (es decir los caracteres del telex), en forma permanente. La falta de precisión en este punto es una fuente de errores.

Tomemos la descripción de los caracteres que intercambian los procesadores de la Central Telex, según las ideas de 1978:

"La comunicación entre Procesadores Centrales y Procesadores Periféricos intercambia octetos que pueden ser caracteres Baudot, órdenes o mensajes. Los caracteres Baudot poseen la distribución de bits:

0 xxxxx 11

El bit mas significativo es nulo. Por el contrario, las ordenes que los Procesadores Centrales envian a los Perifericos poseen siempre el bit mas significativo en 1. Los mensajes poseen ambos formatos."

En esta descripcion solamente se insiste en la estructura de bits de los caracteres Baudot, ordenes o mensajes. Este es un detalle de implementacion, casi sin importancia. No es trivial, en cambio, el hecho que los caracteres Baudot, ordenes o mensajes son muy diferentes de los contadores de tiempos o de la cantidad de caracteres de un texto, de los punteros y de los demas objetos. En la descripcion de 1985, en cambio, el enfasis se pone en la estructura de tipos:

```

package CONSTANTES is
-- caracteres BAUDOT
type BAUDOT is(
    NUL,    -- contenido 00H
    A,      -- contenido 63H
    B,      -- contenido 4FH
    --
    Z,      -- contenido 47H
-- caracteres de control
    LB,     -- contenido 7FH: caracter letras
    CF,     -- contenido 6FH: caracter cifras
    SP,     -- contenido 13H: espacio
    --
-- ordenes hacia el periferico
    OTR,    -- contenido 80H: comenzar comunicacion
    OLL,    -- contenido 81H: llamar
    OCX,    -- contenido 82H: conexion
    --

```

```
);
-- otras definiciones del paquete
end CONSTANTES;
```

La estructura efectiva de bits se encuentra al nivel de un comentario: se ha realizado la abstraccion. Notar que existe sobrecarga en la definicion de los caracteres NUL, SP, A, B, etc. y que este hecho es expresivo porque representan los mismos caracteres graficos que en el alfabeto ASCII.

Tomemos las descripciones de areas segun las ideas de 1978 y 1981. En aquel entonces no existia diferenciacion de tipos y la especificacion de areas era puramente verbal, solamente se ocupaba de las reservas de memoria (tal como le interesa al Assembler). Presentaba este aspecto:

nombre	largo	descripcion
N	-	numero de lineas de la Central: 256
ADE	256	area de entrada de datos de los perifericos
ADS	256	area de salida de datos hacia perifericos
...		
CDOC	256	cola de mensajes de documentacion
CMENS	256	cola de mensajes a consola
...		
LDE	514	lista de estados de lineas
...		
FTEMP	1	bandera de temperatura
...		
TIMER	2	contador BCD de decimas de minuto
...		
XD	1	puntero de salida de CDOC

En lo que sigue se presenta la version de la misma estructura de datos, pero ahora bajo la forma de un paquete ADA:

```

package DATOS is
    --
    N      : constant integer := 256;      -- numero de lineas
    --
    ADE    : array(0..N-1) of BAUDOT;     -- area de entrada
    ADS    : array(0..N-1) of BAUDOT;     -- area de salida
    --
    CDOC   : array(0..255) of character;  -- cola de documentacion
    CMENS  : array(0..255) of BAUDOT;     -- cola de mensajes
    --
    FTEMP  : boolean := FALSE;           -- alarma de temperatura
    --
    LDE    : array(0..N) of ESTADO_DE_LINEA -- lista de estados
              := (0..N-1 => ELB, N => FIN );
    --
    TIMER  : array(0..3) of BCD;         -- decimas de minuto
    --
    XD     : integer range 0..255 := 0;  -- puntero de CDOC
    --
end DATOS;
```

Como puede apreciarse, el paquete DATOS no solamente diferencia claramente nombres y tamaños, tal como se requiere para la implementacion en Assembler, sino que tambien identifica tipos y valor inicial en los casos que interese. Resulta muy expresivo conocer la diferencia de contenido de las colas CMENS y CDOC o introducir los tipos BCD (decimal

empaquetado) y ESTADO_DE_LINEA (definido en la Seccion 4).

Si bien el problema de la especificacion de las estructuras de datos es de caracter general, se convierte en un problema agudo cuando la implementacion se realiza con lenguajes que permiten gran libertad. Esto ocurre en el lenguaje C y en los Assembler y un caso tipico es el manejo de punteros. En estos lenguajes se permite una libertad que puede finalizar en una estructuracion deficiente, en muchos errores y en un mantenimiento muy dificil. La especificacion en ADA obliga a un manejo muy cauteloso de los punteros debido a las restricciones fuertes que impone el lenguaje.

4. EJEMPLOS DE ESPECIFICACION DE ALGORITMOS

El empleo de un lenguaje del alto nivel permite definir con mucha precision los algoritmos a emplear. Presentamos aqui algunos casos que son particularmente interesantes en ADA.

Una aplicacion tipica de ADA, como lenguaje con primitivas orientadas al Hardware, consiste en la descripcion de la informacion de Hardware. Los manuales que describen las piezas del equipo suelen estar escritos por ingenieros electronicos y destinados -como consecuencia- a ingenieros electronicos. Para que los ingenieros de formacion en computacion puedan comprender estos manuales es necesario traducirlos. ADA ofrece una manera muy natural de hacerlo. Este es un ejemplo de uso de una memoria no volatil que posea una de las maquinas usadas, donde la descripcion "tipo Hardware" se reemplaza por esta descripcion "tipo Software":

```
AREA_NO_VOLATIL: array(0..1023) of integer;
```

```
for AREA_NO_VOLATIL use 16#8000#;
```

```
function LEER(dir: integer) return integer is
```

```
DATO: integer;
```

```
begin
```

```
  out(16#20#, 1); -- seleccionar la memoria
```

```
  DATO:=AREA_NO_VOLATIL(dir);
```

```
  out(16#20#, 0); -- deseleccionarla
```

```
  return DATO;
```

```
end LEER;
```

```
procedure ESCRIBIR (dir, dato: integer) is
```

En este caso se usa fuertemente el procedimiento out que reemplaza una accion de entrada/salida del microprocesador. La mayoría de los programadores pueden comprender perfectamente como se maneja esta area no volatil pero no pueden comprender el Manual de Hardware. Es interesante observar que se emplea aqui la clausula de representacion de ADA: pocos lenguajes permitirian este nivel de especificacion.

Un tipo de algoritmo que se presenta frecuentemente es la realizacion de una maquina de estados. Presentaremos un caso tipico del proyecto de Central Telex: cada linea recorre una secuencia de estados. La descripcion de los estados de cada linea se realiza mediante un tipo definido por enumeracion. Una version simplificada se especifica por el texto ADA que sigue:

```
type ESTADO_DE_LINEA is(
    EAL    ,-- linea a liberar
    ECA    ,-- linea en comunicacion
    ELB    ,-- linea libre
    --
    FIN    -- estado de salida
);
```

Con esta definicion de estados, la especificacion de las transiciones -problema tipico de especificacion- se convierte en un algoritmo tipo case y en un problema abstracto, independiente de la implementacion. En el caso de la Central Telex el numero de estados es superior a 120. El diagrama de estados simplificado de la version 1978 ocupaba una hoja de 70 x 90 cm y era sumamente dificil de seguir y actualizar. En 1981 no se intento realizar nuevamente el diagrama. Por otra parte, un diagrama de estados solamente permite indicar algunos elementos de las transiciones pero no toda la actividad propia de cada estado. En el ejemplo ADA que sigue se ilustra la tecnica y se muestra, en forma simplificada, la manera de especificar la actividad y la transicion de estados.


```

procedure ATENCION_DE_LINEAS
  (ADE,ADS: BAUDOT, ESTADO: ESTADO_DE_LINEA) is
begin
  case ESTADO is
    when EAL    =>
      case ADE is
        when A..Z =>
          -- actividad
          ESTADO:=ELB;
        when others =>
          raise ERROR_DE_SISTEMA;
      end case;
    when ECA    => -- actividad
    when ELB    => -- actividad
    --
    when FIN    => -- actividad
  end case;
end ATENCION_DE_LINEAS;

```

En este segmento, además de una especificación de una secuencia de estados, que podría escribirse en muchos otros lenguajes, aparece una excepción. Este es uno de los puntos más importantes de ADA como lenguaje de especificación. Contrariamente a lo que se ha sostenido [8], detectar y manejar en forma abstracta las excepciones es un punto fuerte de un diseño y una garantía de buena implementación.

Es interesante estudiar otro ejemplo, una secuencia de comunicación. En la tabla que sigue aparece toda la secuencia de comunicación, de milisegundo en milisegundo, vista por los Procesadores Centrales y por los Procesadores Periféricos, según las especificaciones de 1978 y 1981:

NINTE	CENTRALES		PERIFERICOS		CONTA
	lee	escribe	lee	escribe	
1	MENSAJE	80H	0	MENSAJE	62
2	MENSAJE	LINEA=1	80H	MENSAJE	-17
3	LINEA=1	LINEA=2	LINEA=1	LINEA=1	-16
.					
18	LINEA=16	0	LINEA=16	LINEA=16	-1

Consideramos que esta descripción solamente puede ser comprendida si se la acompaña de una larga explicación verbal. En el documento se decía, por ejemplo, que se empleaban líneas diferentes para la actividad de Periféricos y Centrales porque las acciones de los Periféricos ocurrían antes que las acciones de las Centrales. Aun así pueden quedar muchas dudas. A título de ejemplo, al redactar la especificación precisa, en ADA, se encontró un error en la tabla que no había sido advertido en más de 6 años.

Esta misma descripción se puede realizar mediante un procedimiento ADA muy simple y que cualquier implementador puede comprender. Consideremos el caso del Procesador Central (el caso Periférico es similar):

```

procedure COMUNICACION (NINTE: integer) is
  XIN: integer:=0;      -- numero de linea en entrada
  XOUT: integer:=0;     -- numero de linea en salida
  procedure PERIFERICO_OUT (B: BAUDOT) is separate;
  function PERIFERICO_IN return BAUDOT is separate;
begin
  delay TIEMPO_DE_SEGURIDAD;
  case NINTE is

```

```
when 1 =>
    MENSAJE_1 := PERIFERICO_IN;
    PERIFERICO_OUT (16#80#);
when 2 =>
    MENSAJE_2 := PERIFERICO_IN;
    PERIFERICO_OUT ( ADS(XOUT) );
    XOUT:=XOUT+1;
when 3..17 =>
    ADE(XIN) := PERIFERICO_IN;
    PERIFERICO_OUT ( ADS(XOUT) );
    XIN:=XIN+1;
    XOUT:=XOUT+1;
when 18 =>
    ADE(XIN) := PERIFERICO_IN;
    PERIFERICO_OUT ( NUL );
    XIN:=XIN+1;

end case;

end COMUNICACION;
```

En este ejemplo se trabaja con la función PERIFERICO_IN y el procedimiento PERIFERICO_OUT. Por lo demás, el programa ADA resulta más preciso que la tabla: se señala que existe una demora de tiempos (TIEMPO_DE_SEGURIDAD), se diferencian los dos mensajes del Periferico, se indica cual es el área de entrada y cual es la de salida (ADE y ADS respectivamente) y se detecta un error de especificación. En efecto, la constante 16#80# no es del tipo BAUDOT, debe emplearse, en lugar del valor, el carácter BAUDOT correspondiente, OTR, ver Sección 3.

5. VENTAJAS DE ADA EN PROBLEMAS DE TIEMPO REAL

El ambiente natural de ADA es el Tiempo Real y de allí que sea en la especificación de este tipo de problemas donde

se logra la mayor ventaja frente a los demás métodos de especificación. Una noción básica del Tiempo Real tal como una interrupción presenta un desafío casi imposible para todos los métodos tradicionales de especificación. Consideremos tres descripciones de una interrupción: la descripción verbal, el diagrama y la especificación en ADA. El resultado será claro. Comencemos por la descripción verbal empleada en 1978:

"Las microcomputadoras empleadas en la Central CTA poseen una única interrupción implementada y corresponde a la realizada por el reloj de tiempo real cada 1 milisegundo. De acuerdo con esto, la programación puede ser clasificada en tres áreas:

1. PROGRAMAS DE INICIALIZACION: ocurren al comienzo y son desencadenados por una señal de reset (de hardware) o por el encendido de las fuentes de la microcomputadora. En el caso del periférico existe, además, un reset por programa.

2. PROGRAMAS DE INTERRUPCION: atienden la interrupción, cuando esta habilitada, según los contadores internos del caso. En el caso de la central, las alternativas principales ocurren según el contador N de interrupciones o según la señal BATEA del reloj. En el periférico, las alternativas están reguladas por tres contadores de interrupciones de características diferentes: CONTA, CONT5 y RGPARG.

3. PROGRAMAS PRINCIPALES: son interrumpidos por el reloj de tiempo real y atienden una tarea que puede ser postergada."

Una descripción equivalente se encuentra en el diagrama que se adjunta, tomada de [1].

Finalmente, consideremos la especificación escrita en ADA:

```
package PROCESADOR_CENTRAL is
with DATOS;           -- estructura de datos
with  CONSTANTES;    -- estructura de constantes
with  IO;             -- estructura de entrada/salida

task BACKGROUND;

task FOREGROUND is

entry SINCRONISMO;
```

```

entry MILISEGUNDO;

end FOREGROUND;

end PROCESADOR_CENTRAL ;

```

Esta especificacion no solamente nos remite a los paquetes de CONSTANTES y de DATOS de la Seccion 3 (el paquete IO no se cita en el presente trabajo) sino que establece claramente el numero de tareas y de puntos de encuentro entre ellas. Todo esto sin necesidad de entrar en detalles de implementacion.

ADA permite mucho mas. Consideremos el siguiente cuerpo para el paquete recién definido:

```

package body PROCESADOR_CENTRAL is

  procedure INIC ;      -- inicializacion

  procedure PCOM ;     -- loop principal

  procedure INTE ;    -- nucleo de tiempo real

  task body BACKGROUND is
  begin
    INIC;              -- inicializar
    FOREGROUND.SINCRONISMO; -- sincronizar

    PCOM;              -- operar
  end BACKGROUND;

  task body FOREGROUND is
    -- declaracion de la interrupcion
    for MILISEGUNDO use at 16#38#;

  begin
    -- espera por sincronismo con el background
    accept SINCRONISMO;

    -- operacion de la interrupcion

```

```

        loop
            accept MILISEGUNDO;
            INTE;
        end loop;
    end FOREGROUND;

-- el cuerpo del paquete es vacio
-- se emplea para desencadenar las tareas

begin
    null;

end PROCESADOR_CENTRAL;

```

En este cuerpo se identifican claramente los diferentes segmentos de programación, sus nombres y su interacción. El punto de "rendezvous" SINCRONISMO permite activar el sistema de interrupciones. La interrupción MILISEGUNDO -cuya rutina de atención se encuentra en la dirección 38 hexadecimal- desencadena el procedimiento INTE. Como podemos apreciar, sobre este paquete que contiene unas pocas líneas, se puede continuar la especificación, módulo a módulo, de arriba hacia abajo, de una manera natural, mucho más precisa que cualquiera de los métodos citados anteriormente y con prescindencia de los detalles de implementación.

El problema considerado, sin embargo, era muy simple. Una única interrupción y un esquema de sincronización elemental. En el ejemplo que sigue se pone de manifiesto toda la potencia formal de ADA para manejar problemas de tiempo real. La siguiente es una descripción de las rutinas de atención de las líneas de la Central Telex:

```

package PAL is
    task type E ; -- atencion de lineas
    RUTINA_DE_LINEA: array(0..N-1) of E ;
end PAL;

package body PAL is
    null; -- entran en actividad las tareas

```

end PAL;

En este fragmento, de una brevedad insuperable, se declara que existen N replicas -una por línea- de una tarea generica E y que todas estas tareas entran en actividad simultaneamente. Por supuesto, se omite la especificacion de E que no es nada simple, pero este fragmento revela toda la estructura que es necesario implementar, independiente de los detalles superfluos.

6. VENTAJAS DEL EMPAQUETAMIENTO

La posibilidad de estructurar la especificacion en paquetes se traduce, en los hechos, en la correspondiente estructuracion de la implementacion. Por esta razon el diseñador debe usar fuertemente esta posibilidad.

Los usos de los paquetes van mas lejos, sin embargo. En los casos que se emplee mas de un lenguaje de programacion, una division adecuada de paquetes permite realizar una division adecuada de lenguajes de programacion. Mas aun, si se implementa la frontera entre ellos, la comunicacion es posible en forma directa.

Un caso muy importante es el empleo de simuladores que permiten ensayar la programacion sin tener completo el sistema. En estos casos basta con invocar un paquete desde un programa que realiza la comunicacion con el operador. De esta manera se puede simular una parte compleja de la programacion. Asi por ejemplo, la secuencia de estados de línea -que posee varios cientos de estados- puede ser simulada mediante un programa ADA como el que sigue:

```

procedure SIMULADOR_DE_LINEAS is
  -- declaracion de constantes y variables
  N          : constant integer :=4 ; -- solo 4 lineas
  ADE, ADS   : array(0..N-1) of BAUDOT;
  LDE        : array(0..N-1) of ESTADO_DE_LINEA;
               := ( 0..N-1 => ELB );
  function GET_BAUDOT return BAUDOT is separate;
begin
  -- mensaje inicial

```

```
loop
    -- ingreso de la nueva entrada
    for l in 0..N-1 loop
        ADE(l) := GET_BAUDOT;
    end loop;
    -- generar una nueva salida
    ATENCION_DE_LINEAS (ADE(1), ADS(1), LDE(1));
    -- escribirla
end loop;
end SIMULADOR_DE_LINEAS;
```

Este programa invoca el procedimiento **ATENCION_DE_LINEAS** que posee, a su vez, una especificacion ADA presentada en la Seccion 4. Nada impide que, una vez realizada la implementacion definitiva en el lenguaje elegido, se construya la comunicacion con el lenguaje ADA y se proceda a trabajar con el simulador, ahora en la implementacion final. De esta manera se pueden detectar errores por comparacion entre la especificacion y la implementacion final. Esta es una herramienta de mucha utilidad en el desarrollo de la programacion.

7. IMPLEMENTACION DE LA ESPECIFICACION

Con la tecnica propuesta la implementacion se convierte en una forma de "compilacion" de la especificacion, pero realizada en forma no algoritmica. Los implementadores actuan como "compiladores inteligentes" y obtienen asi un producto muy superior al que obtendria un compilador. Como es natural, el esfuerzo humano invertido se traduce en un codigo final mas eficiente en tiempo, en memoria o en el empleo de cualquiera de los recursos criticos que se desee optimizar.

A la inversa, en algunos casos fue necesario ir desde "abajo hacia arriba" y traducir la implementacion en un programa ADA para poder verificar su correccion. Los sistemas de despacho de tareas y otros aspectos de mutua exclusion en un nucleo de Tiempo Real requieren demostrar que son correctos, no alcanza con el simple ensayo.

La experiencia nos muestra que la tesis de demostrar la validez de los programas se convierte en una verdadera necesidad en ciertos aspectos de la programacion de tiempo

real donde intervienen problemas de concurrencia. Tambien aqui ADA es una ayuda importante para abstraer los aspectos esenciales del problema.

8. BALANCE DE LA EXPERIENCIA REALIZADA

La experiencia realizada parte de la idea de la necesidad de una especificacion formal para los problemas de informatica tal como Hoare ha expresado tan bien:

"This activity will culminate in a complete, unambiguous, and provable consistent specification for the entire product." [11]

El resultado muestra que, ademas de esta finalidad teorica y posiblemente todavia lejana, aparecen resultados practicos en mas de un aspecto.

La especificacion realizada en ADA mejora claramente la programacion final a pesar de saber que no se emplearon todas las posibilidades del lenguaje. En el caso considerado, los lenguajes de implementacion fueron C y Assembler y el resultado obtenido tiene un estilo de programacion claramente superior a los resultados obtenidos por metodos no formales de especificacion. Sin duda aqui hay una apreciacion que es parcialmente subjetiva pero compartida por todos los miembros del equipo de trabajo.

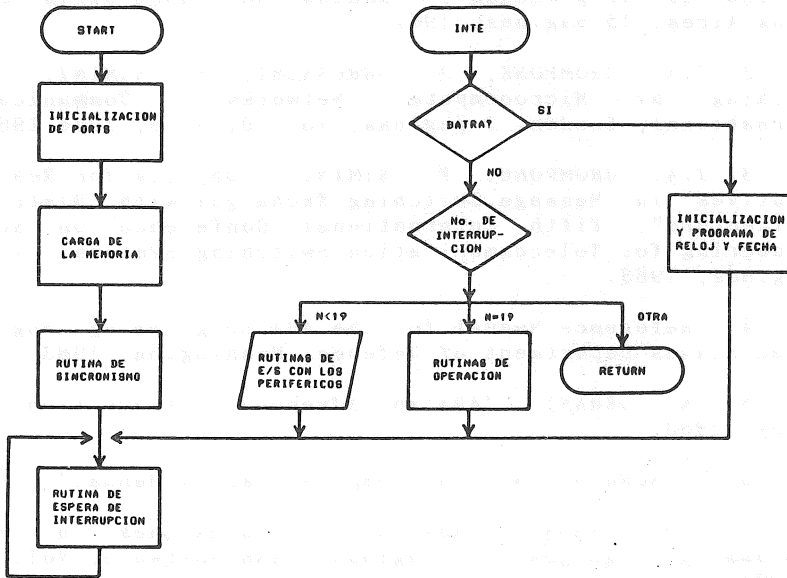
En segundo lugar, una especificacion formal es una manera de comunicar entre si los miembros de un equipo de implementacion de un proyecto, sobre todo si este equipo es grande y de orientacion diferente. En nuestro caso se debian conciliar aspectos de Hardware y de Software con personas de formacion en ambas areas. ADA fue un buen punto de encuentro porque si bien es un lenguaje de alto nivel permite expresar claramente, y en el mismo lenguaje, ambos extremos.

En tercer lugar, la experiencia de especificacion en ADA permitio una activa deteccion de errores. Por un lado, se detectaron errores viejos que llevaban varios años y que no habian sido advertidos. Por otro lado, se tiene la conviccion que todos los errores importantes fueron detectados al redactar el documento de especificacion. La prueba de este hecho es que grandes sub-sistemas resultaron sin errores. Los errores encontrados en la implementacion provenian, casi siempre, de errores en descripciones verbales de documentos relacionados con el proyecto.

Sin duda se dara un enorme paso adelante en la Ingenieria del Software cuando los documentos que debieran ser formales realmente lo sean: manuales de componentes, manuales del sistema operativo, manuales de los compiladores

de los lenguajes, etc. quienes fueron la principal fuente de errores de la experiencia.

Estos resultados de utilidad inmediata justifican que en proximos proyectos se continue con la experiencia de desarrollar especificaciones formales escritas en ADA. Es opinion del autor que este es el camino para adquirir la capacidad necesaria en este complejo aspecto de la ingenieria del Software. Cuando la actividad de especificar formalmente sea una practica corriente y bien conocida, recien se podra avanzar hacia la meta propuesta de demostrar la correccion de las especificaciones y de las implementaciones de la programacion. A pesar de que los resultados obtenidos son muy alentadores, tambien resulta claro que existe un largo camino de desarrollo tecnologico por delante.



Programas del Procesador Central

BIBLIOGRAFIA

- [1] J.A. GROMPONE, N.M.MACE. "Una Central Telex con Procesadores Triplicados", Anales de PANEL'81/12 JAIIO, Buenos Aires, 15 paginas, 1981.
- [2] J.A. GROMPONE, J. JERUSALMI, F. SIMINI. "Telex Switching by Microcomputer Networks", Communications International, London, 3 paginas, Vol 10, N. 6, June 1983.
- [3] J.A. GROMPONE, F. SIMINI. "Objects for Real Time Executives in Message Switching Exchanges with Distributed Architecture", Fifth International Conference on Software Engineering for Telecommunication Switching Systems, London, 5 paginas, 1983.
- [4] "Reference Manual for the ADA Programming Language", United States Department of Defense, Washington, 1983.
- [5] N. GEHANI. "ADA an Advanced Introduction", New Jersey, 1983.
- [6] D. NORRIS. "A (Ada) Compiler User's Manual", 1984.
- [7] M. SHAW. "Abstraction Techniques in Modern Programming Languages", 16 paginas, IEEE Software, Vol. 1 No. 4, 1984.
- [8] C.A.R. HOARE. "The Emperor's Old Clothes", 1980 ACM Turing Award Lecture, Communications of the ACM, 9 paginas, Vol. 24, No. 2, 1981.
- [9] B.A. WICHMANN. "Is ADA Too Big? A Designer Answers the Critics", Communications of the ACM, 6 paginas, Vol. 27 No. 2, 1984.
- [10] J. GROMPONE, F. SIMINI. "An Analysis of Errors Detected During the Design and Implementation of a Real Time Multiprocessor system", enviado para publicar en el IEEE Transactions on Computers, 1985.
- [11] C.A.R. HOARE. "Programming: Sorcery or Science?", 14 paginas, IEEE Software, Vol. 1, No. 2, 1984.