DERIVACION DE CODIGO DE CLASES A PARTIR DE ESPECIFICACIONES FORMALES

Eugenio Scalise, Amelia Soriano y Nancy Zambrano

Laboratorio de Métodos Formales en Ingeniería de Software
Escuela de Computación, Facultad de Ciencias. Universidad Central de Venezuela.
Apartado 47002. Los Chaguaramos, 1041-A. Caracas, Venezuela.
E-Mail: escalise@internet.ve asoriano@kuaimare.ciens.ucv.ve nzambran@strix.ciens.ucv.ve

RESUMEN

En el contexto de un método de construcción de programas por transformaciones a partir de especificaciones formales se presenta un proceso de derivación del código de una clase partiendo del tipo que especifica su comportamiento. Este método permite definir un proceso sistemático y por ende automatizable para generar la clase que implementa al tipo. El código de cada operación se obtiene a partir de su especificación definida mediante una precondición y una postcondición, las cuales son expresadas en términos de la representación seleccionada para el tipo (estructuras de datos o tipos ya implementados y disponibles en una librería). Esta especificación denominada especificación de clase puede considerarse como un primer refinamiento de la implementación. El método cubre varias etapas, este artículo se centra en la etapa de generación de código.

Palabras claves: Interfaz de tipo - Especificación de clase - Especificación algebraica - Generación de código

1 INTRODUCCION

El objeto de este artículo es presentar un método para la derivación del código de una clase o de un paquete expresado en un lenguaje de programación imperativo que provea tales unidades de encapsulación. Este código es generado a partir de la especificación formal del tipo que describe el comportamiento de la clase (las definiciones de tipo y clase se corresponden a los conceptos formulados en UML [4]). El formalismo de especificación de clases ha sido presentado en [9]. Este proceso ha sido automatizado y se ha desarrollado un prototipo (SCAPrI: Sistema de Construcción Automática de Programas Imperativos) [7, 8] que ha permitido probar el método con diferentes casos de estudio para el caso del Lenguaje Ada.

Los datos iniciales del proceso son la *interfaz del tipo* y la *especificación algebraica* asociada. La interfaz del tipo está conformada por las signaturas de las operaciones, cada una definida mediante un encabezado (donde se introduce el nombre de la operación y sus parámetros y tipos asociados, indicando si la operación modifica los datos o los accesa sin modificarlos). A partir de estos datos iniciales se genera una especificación, que se ha denominado *especificación de clase* (en el contexto de los lenguajes de programación orientados a objeto) donde se especifica cada operación asociando a su signatura una precondición y una postcondición.

La especificación algebraica del tipo es una especificación de implementación abstracta la cual especifica el tipo que se desea implementar en función de una representación seleccionada [2]; las pre y post-condiciones se obtienen mediante transformaciones de los axiomas de la especificación algebraica inicial aplicando un proceso sistemático; en este trabajo se omite la descripción de este proceso (presentado en [1]) para centrarlo en el problema de generación de código, donde a partir de los términos algebraicos de la pre y post-condición de cada operación se obtiene el código asociado, el cual puede ser expresado tanto en un pseudo-lenguaje como en un lenguaje de programación destino seleccionado.

2 APLICACIÓN DEL METODO: UN CASO DE ESTUDIO

En esta sección se ilustra mediante un ejemplo las diferentes etapas del método de construcción de programas, mostrando las entradas y salidas de cada etapa. El ejemplo seleccionado corresponde a la implementación del tipo lista en la cual una lista se representa mediante un par formado por un arreglo y un natural. En la sección 3 se describe el proceso de derivación de código.

La generación de la especificación de clase

Para nuestro ejemplo, los datos de entrada son: la interfaz del tipo lista de la figura 1 y la especificación algebraica de la implementación abstracta de listas en funcion de los pares formados por un arreglo y un natural (que constituye el tipo de representación) denominada LIST_AN, la cual se muestra en la figura 2.

Como se puede observar en la figura 1, la signatura muestra para cada operación su nombre y sus parámetros. En el caso que la operación modifique un parámetro ello se indica mediante la palabra clave "mod" antecediendo la variable a modificar (cuyo tipo es el tipo de interés), restringiéndose a la existencia de sólo una variable a modificar. A este tipo de operación se le denominará *modificadora*, y en el caso que acceda los datos sin modificarlos retornando un resultado se le llamará *función*.

```
type name: LIST;
  types: NAT, BOOL;
  operations:
    EMPTY_LIST() return LIST;
    ADD(X: NAT; mod L: LIST);
    IS_EMPTY_LIST(L: LIST) return BOOL;
    FIRST(L: LIST) return NAT;
    TAIL(mod L: LIST);
    BELONG(L:LIST; X: NAT) return BOOL;
```

Figura 1.- Interfaz del tipo de interés LIST

La figura 2 muestra la especificación algebraica de la implementación abstracta de listas llamada LIST_AN, que constituye la *especificación de referencia* relativa a la especificación de clase del mismo tipo. Esta especificación describe las propiedades abstractas que caracterizan la implementación de listas en función de otra abstracción que se supone especificada e implementada (el par formado por un arreglo y un natural). La especificación algebraica del tipo a implementar debe cumplir ciertas hipótesis como estar provista de *constructores*, así como de *selectores* que permitan obtener los componentes del constructor y de *predicados* que permitan reconocer al constructor; la especificación sigue el estilo *constructivo* donde cada operación no constructora está dotada de una *definición completa* en relación a los constructores.

En esta especificación (figura 2) escrita en lenguaje de especificación PLUSS [3] se indica: el nombre de la especificación (cláusula *spec*), la importación de módulos de especificación correspondientes a tipos predefinidos (cláusula *use*) y el sort de interés (cláusula *sort*). En la cláusula *generated by* se indican los constructores. Como puede observarse en la figura 2, la definición del constructor (dado por la operación de abstracción <_,_>) establece que una lista será implementada por un par formado por un arreglo y un natural; las operaciones "ar" e "ind" son los selectores asociados al constructor (obtienen sus componentes); los axiomas de las operaciones de listas se establecen en función de la representación seleccionada. Los tipos predefinidos (ARRAY, NAT y BOOL en este caso) se suponen especificados e implementados.

En la figura 3 se muestra el módulo de especificación de clase generado a partir de la especificación algebraica de referencia asociada al tipo lista (figura 2) y la interfaz de dicho tipo (figura 1). Este módulo contiene el nombre de la especificación de clase (cláusula *class-spec*), el tipo de interés (cláusula *type name*), el nombre de los tipos predefinidos (cláusula *types*). La especificación algebraica de referencia se indica en la clásula *ref-spec*. La cláusula *inherits* indica el nombre de la especificación del tipo de representación utilizado para la implementación abstracta.

```
spec: LIST_AN;
use: ARRAY, NAT, BOOL;
sort: list;
generated by:
  <_,_>: array nat  -> list;
operations:
  ar: list -> array;
  ind: list -> nat;
  empty_list:  -> list;
  add: nat list -> list
  is_empty_list: list -> bool;
  first: list -> nat;
  tail: list -> list;
  belong: list nat -> bool
preconditions:
  first(<t,i>) is-defined iff  not(eq(i, 0) = true;
  tail(<t,i>) is-defined iff  not(eq(i, 0)) = true;
axioms:
  ar(<t, i>) = t;
  ind(<t, i>) = i;
  empty_list = <init(lwb,upb), 0>;
  add(x,<t, i>) = <assign(t.i,x), suc(i)>;
  first(<t, i>) = acces(t, i);
  is_empty_list(<t, i>) = eq(i, 0);
  tail(<t, i>) = <t,suc(i)>;
  is_empty_list(<t,i>) => belong(<t, i>, x) = false
  not(is_empty_list(<t,i>)) => belong(<t, i>, x) =
          if eq(x,first(<t,i>)
          then true
          else belong(tail(<t,i>,x))
where:
  l: list; t: array; i,  x: nat;
end LIST_AN
```

Figura 2.- Especificación Algebraica LIST_AN

```
class-spec: LIST_AN_OP;
  type name: LIST;
  types: ARRAY, NAT, BOOL;
  ref-spec: LIST_AN;
  inherits: RECORD_AN;
operations:
EMPTY_LIST return LIST;
  post: result: <init(lwb,upb), 0>

ADD(X: NAT; mod L: LIST);
  post: l'=<assign(ar(l), ind(l),x), suc(ind(l))>

IS_EMPTY_LIST (L: LIST)return BOOL;
  post: result: eq(ind(L), 0)

FIRST(L: LIST) return NAT;
  pre: not(eq(ind(l), 0))
  post: result: acces(ar(l), pred(ind(l)))

TAIL(mod L: LIST);
  pre: not(eq(ind(l), 0))
  post: l' = <ar(l), pred(ind(l))>

BELONG(L:LIST; X: NAT) return BOOL;
  post: result: if is_empty_list(l)
                then false
                else  if eq(x, first(l)
                      then true
                      else belong(tail(l),x))
end.
```

Figura 3.- Módulo de especificación de clase LIST_AN_OP

En la figura 3, el encabezado de cada una de las operaciones proviene de la interfaz del tipo; las pre y post-condiciones son derivadas de los axiomas de las operaciones en la especificación de referencia aplicando una serie de transformaciones para las cuales se garantiza su correctitud [1]. Los términos de la pre y post-condición están constituidos por las operaciones de las especificaciones predefinidas y del tipo de representación; por hipótesis, estos tipos se encuentran implementados, de allí que estas pre y post-condiciones son "implementables", lo cual es una condición para generar el código de cada operación. Por esta razón, esta especificación puede considerarse como el primer refinamiento de la implementación.

La implementación de la clase: los módulos de implementación del tipo de interés

El proceso de generación del código se realiza a partir de la especificación de clase y requiere adicionalmente como entrada las interfaces de los tipos indicados en las cláusulas *types* e *inherits*. La implementación del tipo de interés puede ser expresada en el pseudo-lenguaje y/o en el lenguaje de programación Ada. A continuación se muestran estos resultados.

El módulo de implementación que se muestra en la figura 4 (expresado en el pseudo-lenguaje) se obtiene a partir de la especificación clase del tipo de interés (figura 3), de las interfaces de tipo de ARRAY, NAT y BOOL y de la interfaz del tipo de representación RECORD_AN (los cuales se suponen disponibles en una

librería), esta información permite determinar si las operaciones requeridas son modificadoras o retornan un resultado (funciones).

El módulo de implementación de la figura 4 contiene el pseudo-código de cada una de las operaciones a ser implementadas. Además, posee un conjunto de cláusulas que indican: el nombre del tipo de interés (cláusula *type name*), los tipos auxiliares utilizados (cláusula *types*), importación de otros módulos (cláusula *use*) y una construcción de equivalencia de tipos mediante la cual se indica que el tipo de interés es equivalente al tipo de representación, heredando sus operaciones y conjunto de valores; en el ejemplo, esta cláusula indica que el tipo LIST es equivalente al tipo RECORD_AN.

La figura 5 muestra el paquete Ada correspondiente a la implementación del tipo de interés. Para la generación de este paquete se requiere del módulo de implementación en el pseudo-lenguaje (figura 4) y la información almacenada en la librería concerniente a las reglas de correspondencia entre las operaciones de los tipos predefinidos y las primitivas en el lenguaje de programación Ada. Así, para obtener el código Ada de una operación es necesario reescribir el pseudo-código en la sintaxis de Ada, tomando en cuenta las reglas de correspondencia.

```
impl-module LIST_AN_M;
    type name: LIST;
    types: ARRAY, NAT, BOOL;
    use: RECORD_AN_M;
    type LIST = RECORD_AN;
begin

EMPTY_LIST() return LIST;
begin
    return(<INIT(LWB,UPB), 0>);
end;

ADD(X: NAT; mod L: LIST);
var T: ARRAY;
begin
    T := AR(L);
    ASSIGN(T, IND(L), X);
    L := <T, SUC(IND(L))>;
end;

IS_EMPTY_LIST(L: LIST) return BOOL;
begin
    return (EQ(IND(L), 0));
end;

FIRST(L: LIST) return NAT;
begin
    if NOT(EQ(IND(L))
    then return(ACCES(AR(L), PRED(IND(L))));
    else FIRST_ERROR;
end;
..
end M_LIST_AN;
```

Figura 4.- Módulo de Implementación LIST_AN_M

```
with ARRAY_LP; use ARRAY_LP;
with NAT_LP; use NAT_LP;
with BOOL_LP; use BOOL_LP;
with RECORD_AN_LP; use RECORD_AN_LP;
package LIST_AN_LP is
   type LIST is private;
   function EMPTY_LIST( ) return LIST;
   procedure ADD(X: NAT; L:  in out LIST );
   function FIRST(L: LIST) return NAT; .....
   FIRST_ERROR, TAIL_ERROR: exception;
private
   type LIST is new RECORD_AN;
end LIST_AN_LP;
package body LIST_AN_LP is
function EMPTY_LIST( ) return LIST is
    T: T_ARRAY;
    begin
        return(T, 0) ;
    end EMPTY_LIST;
procedure ADD(X: NAT; L:  in out LIST) is
    T: T_ARRAY;
    begin
        T := L.AR;
        T(L.IND) := X;
        L := (T, L.IND+1);
    end ADD;
function FIRST(L: LIST) return NAT is
     begin
         if not(L.IND = 0)
         then return(L.AR(L.IND-1));
         else raise FIRST_ERROR;
         end if;
     end FIRST; .....
end LIST_AN_LP;
```

Figura 5.- Paquete Ada LIST_AN_LP

El paquete de implementación Ada contiene instrucciones para la importación de paquetes que implementan los tipos auxiliares y del tipo de representación. En el paquete de especificación se tiene la definición del tipo de interés como un tipo privado, junto con los encabezados de las operaciones implementadas, las cuales son públicas. En el cuerpo del paquete se tiene la implementación Ada de cada una de las operaciones del tipo de interés.

En la parte privada se establece la representación del tipo lista, definiéndolo como un subtipo del tipo RECORD_AN, cuya declaración se encuentra en el paquete importado RECORD_AN_LP mostrado en la figura 6, este paquete establece la definición del tipo de representación y éste es construido a partir de la especificación algebraica del tipo registro (RECORD) realizando una instanciación y un renombramiento.

```
with NAT_LP; use NAT_LP;
  with ARRAY_LP; use ARRAY_LP;
  package RECORD_AN_LP is
    type RECORD_AN is  record
                              AR: T_ARRAY;
                              IND: NAT;
                            end record;
end RECORD_AN_LP;
```

Figura 6.- Paquete de definición en Ada del tipo predefinido RECORD_AN

```
with NAT_LP; use NAT_LP;
package ARRAY_LP is
    LWB := constant 0;        -- Constantes Indicadas por el
    UPB := constant 1000;     -- Usuario (junto con el tipo base)
    type T_ARRAY is new array(LWB..UPB) of NAT;
end ARRAY_LP;
```

Figura 7.- Paquete de definición en Ada del tipo predefinido T_ARRAY

En la próxima sección se muestran los lineamientos generales del proceso de obtención del código de una operación, partiendo de su especificación de clase.

3 PROCESO DE DERIVACION DE CODIGO

En esta sección se describe el proceso general para la obtención del pseudo-código de una operación (que designaremos OP en lo sucesivo), a partir de su especificación de clase y de las signaturas de las operaciones que aparecen en la pre y post-condición de OP. En general, el proceso consiste en convertir los términos de la pre y post-condición de cada operación a código imperativo.

Dependiendo de la naturaleza de la operación OP, se tienen dos casos:

- Si la operación OP es una operación parcial y *tp* es el término de la precondición y *t* es el término de la postcondición, el código generado es de la forma:

  if trad(*tp*) *then* trad(*t*) *else* OP_ERROR

  donde trad(*tp*) y trad(*t*) indican los códigos secuenciales obtenidos respectivamente de la traducción de los términos *tp* y *t*; OP_ERROR corresponde a un procedimiento de error que se ejecuta cuando la precondición de OP no se cumple, finalizando la ejecución del programa. En el ejemplo, éste es el caso de las operaciones FIRST y TAIL (figura 4).

- Si OP es una operación total (no existe precondición), el código se deriva directamente del término *t* de la postcondición, en cuyo caso trad(*t*) indica el código secuencial obtenido del término *t*. En el ejemplo, éste es el caso de la operación EMPTY_LIST, ADD y BELONG (figura 4).

El principio general para la derivación del código a partir de un término consiste en el tratamiento de los sub-términos funcionales de izquierda a derecha y del más interno al más externo, en caso de existir términos anidados; adicionalmente se consideran las signaturas de las operaciones que figuran en su pre y post-condición (provenientes de las interfaces de los tipos predefinidos y del tipo de representación) a fin de determinar el carácter de cada operación (función o modificadora) al momento de derivar el código imperativo.

A continuación se describe el corazón del algoritmo de generación de código a partir de un término simple de la forma $t = f(t_1, t_2,...,t_n)$ donde cada subtérmino t_i puede ser una variable, una constante o un término

equivalente, esto es $t_i = f_i(t_{i1},..,t_{im})$. Por simplificación se considera el término simple a traducir de la forma: $t = f(f_1(...),f_2(...),...,f_n(...))$. Para generar el código asociado, para cada sub-término de raíz f_i de f se debe:

- Derivar el pseudo-código de cada uno de los sub-términos de f_i , obteniendo: <lista-inst $t_{i1},...,t_{im}$>
- Derivar el código del término $f_i(...)$, obteniendo $F_i(...)$
- Según el carácter de F y F_i se tienen varios casos para el código del término de entrada t:

1. Si F_i es una operación modificadora (cuya variable "mod" ocupa la posición k) y F una función o una operación modificadora, el código del término t es:

   <lista-inst $t_1,...,t_{i-1}$>; <lista-inst $t_{i1},...,t_{im}$>; $F_i(...,\underline{X}_k,...)$; <lista-inst $t_{i+1},...,t_n$>; [] $F(...,\underline{X}_k,...)$; []

   {donde $\underline{X}_k$ es el i-ésimo argumento de F}

2. Si F_i es una función y F una función o una operación modificadora cuyo i-ésimo argumento corresponde a un parámetro por valor, entonces el código del término t es:

   <lista-inst $t_1,...,t_{i-1}$>; <lista-inst $t_{i1},...,t_{im}$>; <lista-inst $t_{i+1},...,t_n$>; [] $F(...,F_i(...),...)$; []

3. Si F_i es una función y F es una operación modificadora cuyo i-ésimo argumento corresponde a su variable 'mod', entonces el código del término t es de la forma:

   <lista-inst $t_1,...,t_{i-1}$>; <lista-inst $t_{i1},...,t_{im}$>; $\underline{X}$:= $F_i(...)$; <lista-inst $t_{i+1},...,t_n$>; $F(...,\underline{X},...)$; []

La variable subrayada indica que ésta puede ser una variable argumento de OP o una nueva variable que eventualmente puede añadirse en el pseudo-código producido, lo cual es determinado estudiando el carácter de la operación OP. El símbolo [] indica que el pseudo-código puede completarse dependiendo del carácter de F y de OP, ya sea con una instrucción "return F(...);" si OP es una función o con una asignación sobre la variable 'mod' de la operación OP (en caso que OP sea modificadora), ésto se logra añadiendo una asignación al código producido de la forma "<nombre-variable-mod> := <resultado>" si es necesario.

El código producido adquiere características imperativas al contener llamadas a operaciones implementadas como modificadoras y al poseer eventuales asignaciones sobre variables. Estas asignaciones se hacen presentes en el código final cuando debe almacenarse el resultado de una función o cuando se realizan copias de variables.

El algoritmo de derivación del código de una operación

A continuación se describe la estructuración del algoritmo de derivación del código de una operación a partir de su especificación de clase. Este algoritmo está centrado en la generación del árbol que representa el programa, su recorrido produce la escritura del código.

Se definen los procesos TRADUCIR, TRAD_TERM_IF y TRAD_TERM_SIMPLE que generan diferentes partes del código de una operación, de acuerdo al término de entrada. En la figura 7 se muestra gráficamente el flujo de datos entre los procesos, cuyo retorno (implícito en la gráfica) constituye el código asociado al término de entrada; a continuación se describen los procesos.

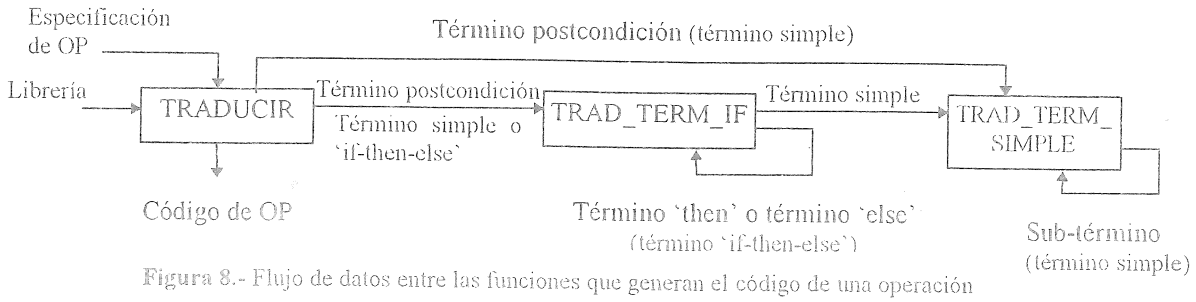

Figura 8.- Flujo de datos entre las funciones que generan el código de una operación

Se recuerda que el término de la precondición es siempre un término simple, mientras que el término de la postcondición puede ser un término 'if-then-else'. Los términos correspondientes a las partes 'then' ('término-then') o 'else' ('término-else') pueden ser términos simples o términos 'if-then-else'.

TRADUCIR

La entrada de este proceso es la especificación de una operación OP y genera como resultado el código de dicha operación (en su ambiente global accede a la especificación de clase correspondiente al tipo de interés y a las interfaces de los tipos involucrados).

El proceso:

- Función que toma la precondición (si ella existe) y la postcondición de la operación a implementar.
- Si OP tiene precondición, TRADUCIR invoca a TRAD_TERM_SIMPLE pasándole el término simple de la precondición como argumento.

  Luego, TRADUCIR invoca a TRAD_TERM_IF pasándole el término de la postcondición.
  Con los resultados retornados, TRADUCIR realiza el ensamblaje de la expresión 'if-then-else' con el código de la precondición, el código de la postcondición (parte 'then') y un código por defecto (parte 'else') que corresponde al caso en que la precondición sea falsa. Si no existe precondición el código resultante corresponde a la traducción del término de la postcondición.
  Finalmente se ensambla la declaración de las variables locales al código para producir el programa final.

TRAD_TERM_IF

- Su objetivo es descomponer el término de la postcondición si es un término 'if-then-else' hasta obtener los términos simples que son pasados a TRAD_TERM_SIMPLE y finalmente realiza el ensamblaje de las partes de código retornadas para formar la expresión 'if-then-else'. Si hay anidamientos de términos 'if-then-else', el primero que es ensamblado es el más interno.
- TRAD_TERM_IF concluye la construcción del código correspondiente al término añadiendo la instrucción 'return' (si la operación OP es una función) o una instrucción de asignación a la variable 'mod' de OP que asegure que el resultado queda en esta variable (si la operación OP es modificadora).

TRAD_TERM_IF produce una lista y un programa que representan respectivamente la lista de variables locales asociada al estado actual del programa y el código parcial asociado al término de entrada.

TRAD_TERM_SIMPLE

- TRAD_TERM_SIMPLE realiza el tratamiento de los términos simples, si hay anidamientos de términos realiza una llamada recursiva y el primer sub-término que se trata es el más interno; una vez generado el código del término simple es retornado a TRAD_TERM_IF o a TRADUCIR.

TRAD_TERM_SIMPLE produce: el código parcial asociado al término simple de entrada y la lista de variables locales asociada al estado actual del programa.

El núcleo de este algoritmo ha sido descrito en la sección precedente.

La combinación de estos procesos conducen a la obtención del árbol que representa el programa.

El algoritmo de derivación de código y las estructuras de datos

Los objetos participantes en el proceso de generación de código, tanto la especificación de clase como el código generado, son representados mediante árboles (forma interna). El árbol abstracto del programa generado es recorrido para escribir el código en forma textual (también llamada forma externa).

Se presenta, mediante ejemplos, la forma externa e interna del código generado al aplicar los algoritmos para la obtención del código de una operación. Se muestra la representación lógica de las estructuras que

definen la forma interna del código producido (se omiten aspectos que obligarían a entrar en detalles del diseño y de la implementación).

1) La implementación de la operación OP

En la figura 9 se muestra las entradas y salidas del proceso de generación de código de la operación OP en su forma externa (forma de texto): la especificación de OP y las signaturas de las operaciones que aparecen en la postcondición (de las interfaces de los tipos predefinidos) así como el código resultante una vez aplicado el proceso de generación.

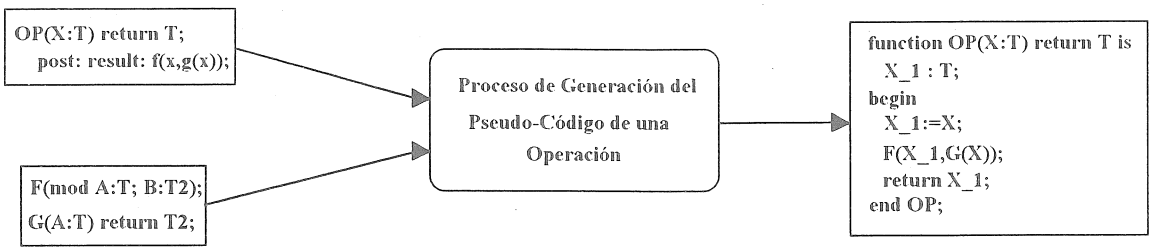

Figura 9.- Entradas y salidas del proceso de generación de código de la operación OP

La representación (forma interna) del código generado es un árbol (figura 10) cuyos nodos ovalados corresponden a construcciones y elementos del pseudo-lenguaje, mientras que los nodos rectangulares corresponden a identificadores de operaciones (funciones o modificadoras) o variables del código generado.

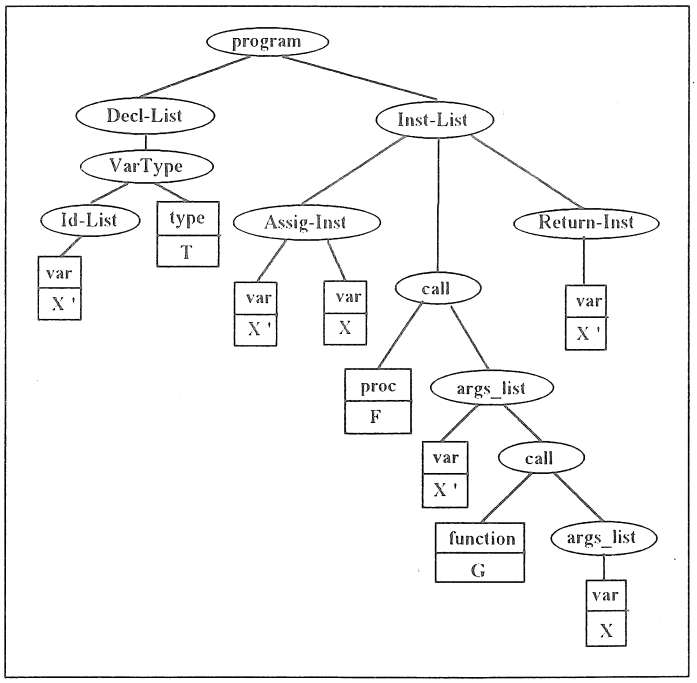

Figura 10.- Forma interna del código generado

El árbol abstracto del programa es recorrido para escribir el código en la sintaxis del pseudo-lenguaje o del lenguaje de programación destino; para este último se aplican las correspondencias entre las construcciones del pseudo-lenguaje y el lenguaje de programación (Ada, en este caso), también se aplican las reglas de correspondencia asociadas a las operaciones de los tipos predefinidos.

2) La implementación de la operación FIRST del tipo LIST

La figura 11 muestra las entradas y salidas del proceso de generación de código para la operación FIRST en su forma externa (texto); como entradas se tiene la especificación de FIRST y las signaturas de las operaciones requeridas (provistas en las interfaces de los tipos RECORD_AN y ARRAY); como salida del proceso de generación de código se muestra el código producido para la operación FIRST.

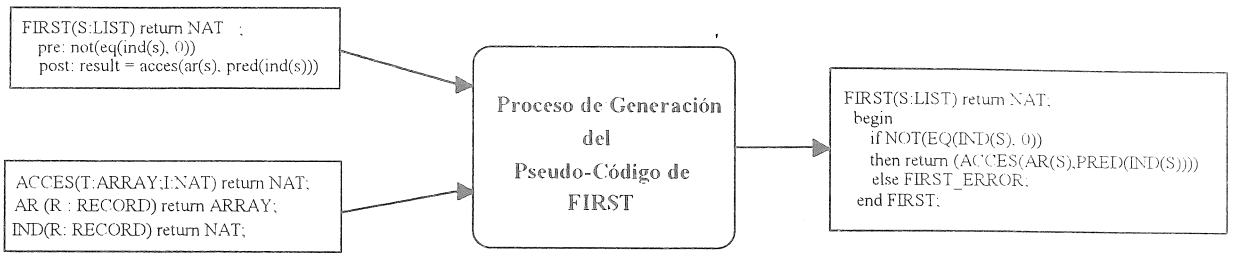

Figura 11.- Entradas y salidas del proceso de generación de código de la operación FIRST

Las figuras 12 y 13 presentan los sub-árboles generados correspondientes al código de los términos de la pre y de la post-condición. El ensamblaje de la instrucción 'if-then-else' es efectuado posteriormente. Así, el código asociado al término de la precondición `not(eq(ind(s),0))`, denotado *cond-code*, es:

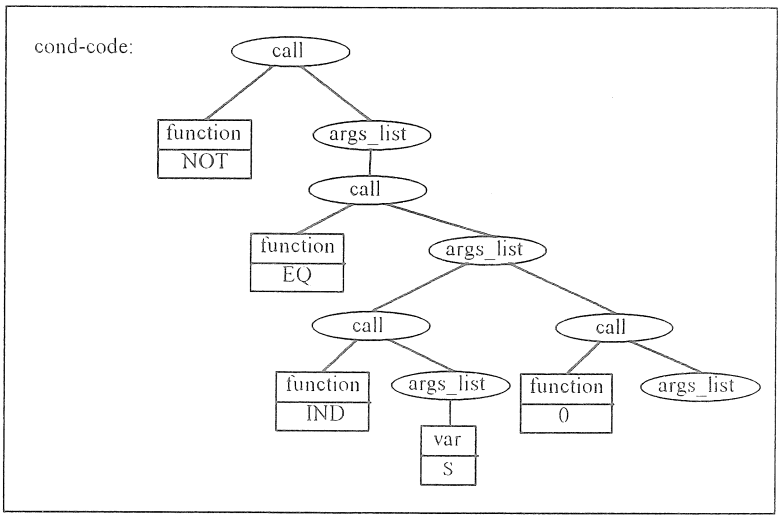

Figura 12.- Forma interna del código generado asociado a la precondición de la operación FIRST

La figura 13 muestra los árboles que representan el código asociado al término de la postcondición dada por el término `acces(ar(s),pred(ind(s))))`.

Puesto que que FIRST es una operación parcial, el código resultante es un condicional de la forma: "*if* trad(*tp*) *then* trad(*t*) *else* FIRST_ERROR" donde *tp* y *t* representan los términos de la precondición y de la postcondición respectivamente. El árbol denotado *then-code* corresponde a la parte 'then' de la instrucción condicional y *else-code* corresponde al código por defecto, asociado a la parte 'else' de la misma instrucción condicional. Finalmente, una vez generados estos subárboles, en TRADUCIR se efectúa el ensamblaje de la expresión condicional, para producir el código asociado a la operación a implementar (FIRST).

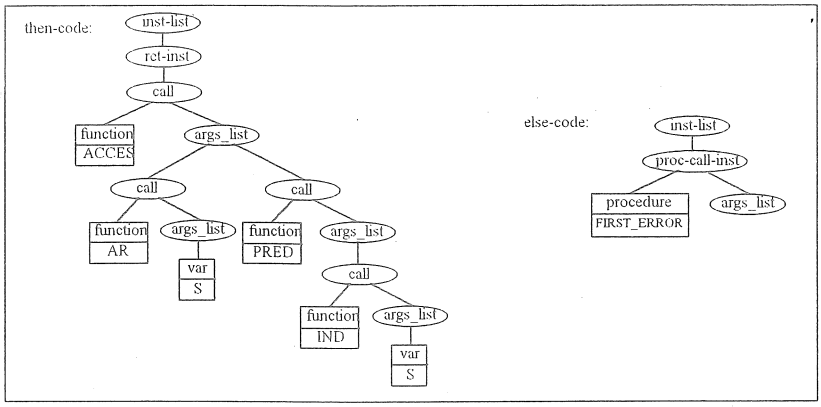

Figura 13.- Forma interna del código asociado a la postcondición y a la excepción de la precondición

4. CONCLUSIONES

El método presentado en este trabajo constituye un proceso sistemático el cual permite obtener en forma automática la clase que implementa un tipo de interés. Esto se logra partiendo de la especificación algebraica del tipo (y de su interfaz), la cual es transformada para obtener la especificación de clase y finalmente generar el código imperativo que implementa al tipo de interés. El código de las operaciones se obtiene aplicando el proceso de derivación descrito a los términos de la pre y post-condición de cada operación. Esto constituye una diferencia fundamental con otros enfoques tales como LARCH [5] y VDM [6], en los cuales las especificaciones con pre y post-condiciones guían al programador, ya sea en las pruebas o para aplicar un proceso manual de refinamiento hasta obtener el código.

Si bien este trabajo se ha centrado en describir el proceso de generación de código a partir de especificaciones formales pueden señalarse algunos consideraciones generales referentes al aspecto de prueba.

Los modelos de desarrollo de software por transformaciones siguen, en general, dos enfoques en lo referente a las pruebas que deben realizarse para garantizar que el programa satisface la especificación inicial: el primero se basa en pruebas que se aplican en cada paso del refinamiento para cada problema; el segundo, en la prueba del proceso, lo cual es requerido sólo una vez y luego se aplica el proceso a cada problema.

El método que se propone sigue el último enfoque, este proceso tiene como entrada la interfaz del tipo a implementar y la especificación algebraica de la implementación abstracta del tipo y como salida la clase que implementa al tipo. Respecto a la especificación algebraica de entrada es necesario señalar:

- Debe garantizarse que la especificación sea consistente y completa. Intuitivamente la completitud corresponde al problema de saber si los axiomas de la especificación son suficientes para describir el sistema. La consistencia responde al problema de saber si los axiomas de la especificación no son contradictorios. Estas propiedades son indecidibles en general y diversos trabajos muestran como el establecimiento de condiciones sintácticas impuestas a las especificaciones aseguran tales propiedades [11]. Es por ello que las especificaciones de entrada deben cumplir ciertas condiciones entre las cuales se destaca que siguen el *estilo constructivo* (o presentación *graciosa* en [11]) en el cual se dota a las operaciones de una definición completa en relación a los constructores. Esta y otras condiciones que deben cumplir las especificaciones de entrada garantizan además que el sistema de reescritura formado por las ecuaciones orientadas es a terminación finita.
- Los aspectos referentes a la correctitud de una *especificación algebraica de implementación abstracta* han sido extensamente estudiados [2]. Los criterios de corrección se expresan mediante conceptos como la inducción estructural o la consistencia jerárquica, ésto es, se basan en el conocimiento de la especificación de la implementación abstracta y se ha demostrado que ella es isomorfa a la especificación algebraica descriptiva que precisa las propiedades abstractas que caracterizan el tipo que se desea implementar.

En lo referente al proceso puede señalarse:

- Para obtener la especificación con pre y post-condiciones se requiere, en primer lugar, pasar del estilo constructivo de la especificación al estilo funcional. Para ello se aplican un conjunto de transformaciones a los axiomas de la especificación inicial y se demuestra que después de cada transformación el axioma obtenido es equivalente al axioma de entrada haciendo uso del razonamiento ecuacional y basándose en conceptos clásicos de la teoría de tipos algebraicos [10].
- Apoyándose en las definiciones de la especificación de clase [9] y utilizando la aplicación de modelación entre la interfaz del tipo y la signatura algebraica, se realiza el pasaje a una especificación con pre y post-condición a partir del axioma expresado en la forma funcional; a manera de ejemplo, si se tiene la definición funcional de una operación total *op*, por ejemplo, $op(x_1,...x_n) = t$, el término *t* será la postcondición de la operación $OP(X_1,...X_n)$, luego se tendrá: post: result = t (o post: x` = t si OP modifica a la variable x).

Para el proceso de generación de código a partir de la pre y post-condición, las transformaciones que se efectúan tienen por objeto realizar el cálculo de la expresión funcional sujeto a las reglas que impone el lenguaje de programación destino. Estas transformaciones se basan entonces en la aplicación de una serie de reglas que interpretan la semántica del lenguaje de programación.

Para ejemplificar, si la postcondición de la operación OP (que se desea implementar en el lenguaje Ada) está dada por el término $t = f(f_1(...), f_2(...),..., f_n(...))$, suponiendo el caso más simple en que todas las operaciones que figuran en el término han sido implementadas como funciones, el código derivado será de la forma: return($F(F_1(...), F_2(...),..., F_N(...))$) si OP es una función. Sin embargo, algunas operaciones F_i (o F) podrían estar definidas como procedimientos y ello introduce la necesidad de transformaciones que llevan a un cálculo secuencial, ya que si una operación F_i es un procedimiento, el argumento que ocupa la posición de su parámetro modificable debe ser una variable, además que esta operación F_i no puede ser un argumento de otra operación. Otras restricciones impuestas por el lenguaje (por ejemplo, los parámetros por valor de la operación OP no pueden ser modificados en su ejecución; etc.) lleva a completar un conjunto de reglas que permiten hacer la traducción. La aplicación de estas reglas permiten obtener el código secuencial corres-pondiente a los términos funcionales de la pre-y post-condición. La verificación puede realizarse mediante testeo o prueba formal asumiendo el conjunto de las reglas de inferencia.

En lo que concierne al algoritmo de derivación de código al aplicar las reglas en forma sistemática puede obtenerse en algunos casos un código no eficiente (por ejemplo, la introducción de instrucciones de asignación que realicen copias innecesarias). Actualmente se investiga sobre la optimización del código generado, sea aplicando técnicas de transformación de programas o considerando pre y post-condiciones que incluyan modificadores de componentes de estructuras complejas, introducidos desde la especificación algebraica misma.

REFERENCIAS

[1] ACOSTA A., ZAMBRANO N. "GEALOP: Transforming Algebraic Specifications". Proceeding ISAS'97, International Conf. on Information Systems Analysis and Synthesis. Caracas, Venezuela, 1997

[2] BERNOT G. "Correctness Proof for Abstract Implementations". Information and Computation. Vol.80(2). 1989.

[3] BIDOIT, Michel. "PLUSS, un langage pour le développement de specifications algébriques modulaires". These Docteur D'ETAT. Université Paris-Sud. Orsay, 1989.

[4] BOOCH, G., RUMBAUGH, J., JACOBSON, I. "UML (Unified Modeling Language)" version 1.0. www.rational.com. 1997.

[5] GUTTAG J. V., HORNING J.J. "LARCH: Languages and Tools for formal specification". Texts and monographs in Computer Science. Springer-Verlag. 1993.

[6] JONES C. B. "VDM. Une méthode rigoureuse pour le développement du logiciel". MASSON, 1993

[7] SCALISE, Eugenio. "SCAPrI: Sistema para la Construcción Automática de Programas Imperativos a partir de especificaciones formales". Trabajo de Grado. Esc. de Computación, Facultad de Ciencias, UCV, 1996.

[8] ZAMBRANO, Nancy. ACOSTA Eleonora. "An Automatic Generation programs environment ". Proceeding ISAS'96 International Conference on Information Systems Analysis and Synthesis. Orlando, USA. 1996.

[9] ZAMBRANO N., SCALISE E., SORIANO A. "Formal specification of classes in the algebraic context". Proc. ISAS'97. International Conf. on Information Systems Analysis and Synthesis. Caracas, 1997

[10] ZAMBRANO N. "Une méthode de dérivation de programmes impératifs à partir de spécifications algébrico-opérationnelles". These Docteur en Sciences. Université Paris-Sud. Orsay, 1995

[11] BIDOIT, Michel. "Algebraic Data Types: Structured Specifications and 'fair' presentations". Proc. AFCET Simposium on Mathematics for Computer Science. 1982.