UN MODELO DE DESARROLLO DE SOFTWARE PARA LA GENERACION AUTOMATICA DE CLASES

Eugenio Scalise, Nancy Zambrano

Laboratorio Métodos Formales en Ingeniería de Software, MeFIS.
Centro ISYS. Locales III. Escuela de Computación, Facultad de Ciencias.
Universidad Central de Venezuela. Apartado 47002.
Los Chaguaramos, 1041-A. Caracas, Venezuela.

e-mail: escalise@acm.org, nzambran@strix.ciens.ucv.ve

RESUMEN

El presente trabajo introduce un modelo de desarrollo de software basado en transformaciones que permite derivar de manera automática clases en lenguajes de programación orientados a objeto (Ada 95, C++, Eiffel y Java) partiendo de especificaciones formales. El conjunto de transformaciones que conforman el modelo de desarrollo presentado constituyen pasos sistemáticos que parten de la especificación algebraica de un tipo. Esta especificación algebraica describe el comportamiento abstracto de un tipo (tipo de interés) en términos de otro previamente especificado e implementado (tipo de representación). Las transformaciones que conforman el modelo permiten de manera progresiva acercar la especificación de partida a un programa (clase) equivalente al tipo de interés inicial. Estas transformaciones permiten obtener en primer lugar una especificación intermedia (especificación de clase) que introduce la noción de estado utilizando pre y post-condiciones para describir las operaciones del tipo de interés. Luego, se aplican transformaciones que permiten obtener código imperativo independiente del lenguaje de programación destino (pseudo-clase), que finalmente es transformado a cualquier lenguaje de programación orientado a objeto para el cual se hayan definido las transformaciones correspondientes.

Palabras claves: modelos de desarrollo de software basado en transformaciones, construcción automática de programas, especificaciones formales, especificaciones algebraicas, especificaciones de clase, tipo de dato, clase, lenguajes orientados a objeto.

1 INTRODUCCION

En este artículo se describe un modelo de desarrollo de software basado en transformaciones cuyo objetivo es obtener la implementación concreta de un tipo cualquiera (clase) partiendo de la especificación algebraica del tipo en términos de una representación abstracta.

Un modelo de desarrollo de software basado en transformaciones tiene como objetivo principal construir un programa P que sea equivalente a una especificación SP dada (o semánticamente: $Mod(P) \subseteq Mod(SP)$, donde Mod(A) representa el conjunto de modelos que validan las sentencias de A). La construcción del programa P a partir de la especificación SP consiste en una secuencia de pasos de refinamiento:

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n$$

En esta cadena, SP_0 es la especificación original, SP_n es el programa P y $SP_{i-1} \rightsquigarrow SP_i$ para i = 1,...,n es un paso de refinamiento. Estos pasos sucesivos garantizan que el programa derivado es equivalente a la especificación original si los pasos de refinamiento han sido probados; en otras palabras, SP_{i-1} se refina como SP_i (denotado $SP_{i-1} \rightsquigarrow SP_i$) si todo modelo de SP_i es un modelo de SP_{i-1}. Estos pasos pueden ser descritos en forma sistemática mediante reglas de transformación tales que los resultados son correctos si las condiciones impuestas sobre las reglas se cumplen. Esto evita probar la correctitud de cada paso de transformación a cada problema, ya que basta únicamente probar la correctitud de las transformaciones y garantizar en cada paso que se satisfacen las condiciones de la misma [7]. Las transformaciones asociadas a cada paso pueden describirse de manera algorítmica o formal, indicando las entradas, sus transformaciones y la salida resultante. Estas ideas de transformaciones sucesivas para la producción automática de código, sustentan el modelo de desarrollo de software que se describe en este artículo.

El modelo de desarrollo de software aquí presentado constituye una extensión de un método para la construcción automática de programas imperativos presentado en [11] el cual fue optimizado e implementado en [8]. Las extensiones principales

consisten en obtener un modelo de desarrollo de software multilenguaje y con soporte a abstracciones de datos genéricas; tomando en cuenta además, que los lenguajes destino son lenguajes orientados a objeto. Los avances iniciales de esta investigación fueron presentados en [10]. Los detalles del modelo de desarrollo de software presentado así como su justificación teórica se encuentran presentados en [9].

En este artículo se describe en primer lugar los objetos que intervienen en el modelo de desarrollo de software, así como las relaciones entre éstos. Luego, se presentan las diversas transformaciones que conforman el modelo, presentadas en dos bloques.

2 OBJETOS DEL MODELO

En esta sección son descritos los diversos objetos involucrados en el modelo de desarrollo de software. Es importante señalar que los conceptos de tipo y clase utilizados en este trabajo se basan en las definiciones presentadas en UML [6]. En este contexto, un *tipo* se define como un descriptor para objetos con estados abstractos y especificación de sus operaciones y una *clase* constituye un descriptor para objetos con estados concretos e implementación de operaciones. De acuerdo a estas definiciones, un tipo especifica una clase y recíprocamente una clase implementa un tipo.

2.1 Interfaz de tipo

Una interfaz de tipo representa la signatura o parte pública de un tipo de datos cualquiera; conformada por: un conjunto finito de tipos (entre los cuales se distingue un tipo especial, denominado *tipo de interés*) y un conjunto de operaciones con su respectivo perfil. El perfil de una operación está representado mediante un encabezado de función u operación modificadora.

La interfaz de un tipo constituye el punto de partida para la obtención de la implementación de un tipo de interés, ya que a través de ésta se indican los aspectos generales del tipo de interés tales como: las operaciones que define el tipo, así como los tipos auxiliares que se requieren. Esta información resulta útil tanto para los usuarios del tipo como para los procesos de transformación, entre los cuales cabe destacar el proceso de derivación automática del código, para el cual la interfaz del tipo define el "protocolo" o la forma en que las operaciones deben ser implementadas (como funciones o modificadoras), además de mostrar la información de sus parámetros formales (orden, tipo, forma de transferencia, etc.).

```
interface name: GEN_SET[ELEM];
type of interest: SET;
types: NAT, BOOL;
operations:
    ...
    DELETE(mod S:SET; E:ELEM);
    ...
end.
```

Fig. 1.- Interfaz de tipo GEN_SET[ELEM]

A manera de ejemplo, en la Figura 1 se presenta la interfaz `GEN_SET[ELEM]` del tipo conjunto de componentes genéricos. La cláusula `interface name` indica el nombre del módulo de interfaz de tipo que puede contener opcionalmente la definición de componentes genéricos, indicados como una lista de nombres entre corchetes. La cláusula `type of interest` indica el nombre del tipo definido por la interfaz (tipo de interés). La cláusula `types` indica los tipos auxiliares necesarios para la definición del tipo de interés. Adicionalmente, existe una cláusula opcional `inherits`, en la que se indica si el tipo de interés definido por la interfaz constituye una especialización de otro tipo, en cuyo caso hereda las operaciones del tipo padre; de esta manera es posible enriquecer el tipo padre e inclusive redefinir operaciones ya existentes. Los encabezados de las operaciones del tipo de interés son agrupados en la cláusula `operations`, diferenciando entre funciones y operaciones modificadoras dependiendo del caso en que la operación tenga tipo de retorno (función) o posea uno y sólo un parámetro modificable (operación modificadora).

2.2 Especificación Algebraica

Una especificación algebraica ALG_{spec} es una tripleta $<S,\Sigma,Ax>$, donde S es un conjunto de sorts, Σ es el conjunto de operaciones y Ax representa el conjunto de axiomas. Los axiomas de la especificación expresan las propiedades abstractas del tipo de datos. En general, los axiomas pueden ser ecuaciones, ecuaciones condicionales, fórmulas expresadas en lógica de primer orden, etc.

Las especificaciones algebraicas pueden ser de dos tipos:

– Una especificación algebraica *descriptiva*, que describe las propiedades abstractas que caracterizan el tipo de datos que se desea definir.

– Una especificación de *implementación abstracta*, que describe las propiedades que caracterizan la implementación del tipo de interés mediante otro tipo que se supone especificado e implementado.

El modelo de desarrollo de software propuesto está basado en especificaciones de implementación abstracta, en particular el enfoque presentado en [2]. Adicionalmente, se seleccionó como lenguaje de especificación un subconjunto del lenguaje PLUSS [3], que permite ensamblar diferentes módulos de especificación para formar especificaciones complejas, estructuradas, modulares y genéricas.

```
generic spec: SET(ELEM);
use: NAT, BOOL;
use: LIST;
sort: set;
generated by:
    <_>: list -> set;
operations:
    ...
    delete: set*elem -> set;
    ...
preconditions:
    delete(<l>,e) is-defined if
                    is_in(l,e) is-true;
axioms:
    ...
    delete(<l>,e)=<remove(l,e)>;
    ...
where:
    l:list; e:elem;
end SET(ELEM).
```

Fig. 2.- Especificación Algebraica SET(ELEM)

En la Figura 2, se presenta como ejemplo una parte de la especificación algebraica genérica `SET(ELEM)`, la cual define el comportamiento abstracto del tipo conjunto (`SET`) en términos de una lista genérica (`LIST`). Como se puede observar en el ejemplo, la cláusula `generic spec` indica el nombre del módulo de especificación algebraica, que en este caso es genérico. La cláusula `use` indica la importación de módulos de especificación de sorts predefinidos y sorts auxiliares. PLUSS permite varias cláusulas `use`, por convención se utiliza una segunda cláusula de este tipo para indicar el nombre del módulo de especificación asociado al sort de representación, estableciendo una diferenciación entre los sorts auxiliares y el sort de representación. La cláusula `sort` permite indicar el nombre del sort de interés definido por la especificación. El resto de las cláusulas permiten definir los perfiles de las operaciones constructoras (`generated by`) y no constructoras (`operations`), las precondiciones asociadas a las operaciones parciales (`preconditions`), los axiomas de las operaciones en la forma de ecuaciones, aserciones o ecuaciones condicionales (`axioms`) y las declaraciones de variables utilizadas en los axiomas (`where`).

2.3 Especificación de Clase

Una especificación de clase es un formalismo de especificación en el cual las operaciones de una clase son especificadas en base a pre y post-condiciones, expresadas en términos de una representación. La precondición expresa las hipótesis relacionadas con los argumentos de las operaciones y define las condiciones que deben satisfacerse para aplicar la operación; la postcondición expresa el efecto o cambio de estado producidos por la operación como consecuencia de su aplicación. Este formalismo es definido en el marco del modelo de desarrollo de software aquí presentado y su definición detallada y aspectos semánticos se encuentran detallados en [9].

Una especificación de clase $\text{CLASS}_{\text{spec}}$ está definida por una tupla $(<\sum_{\text{interface}}, \text{ALG}_{\text{spec}}, R>, \wp)$, donde $\sum_{\text{interface}}$ representa la interfaz del tipo, ALG_{spec} representa la especificación algebraica de referencia, R es una relación entre las signaturas $\text{Sig}(\text{ALG}_{\text{spec}})$ y $\sum_{\text{interface}}$. El conjunto de pre y post-condiciones de las operaciones del conjunto $\sum_{\text{interface}}$ es representado por $\wp$.

Las operaciones de una especificación de clase están conformadas por un encabezado (el cual proviene de la interfaz de tipo), una precondición y una postcondición. Estas últimas constituyen términos funcionales con variables que se derivan de los axiomas de la especificación de referencia, mediante un proceso de transformaciones descrito en la Sección 3.

En general, las operaciones de una especificación de clase pueden ser parciales o totales y dependiendo de esto, el término de la precondición puede ser opcional. El término de la postcondición puede ser de diversas formas dependiendo si la operación es modificadora o función. Si la operación es modificadora se indica en forma explícita la modificación del parámetro por referencia (`mod`, en el contexto del modelo) con el valor del término de la postcondición de esta forma: `post: x'=t` (siendo `x` el parámetro `mod` y `t` el término de la postcondición). En caso contrario se denota como: `post: result=t`.

La especificación de clase es la entrada de un proceso de derivación automática de clases en diversos lenguajes de programación (por ejemplo: Ada 95, C++, Eiffel, Java, etc.), el cual será descrito en la Sección 4.

En la Figura 3 es mostrada una parte de la especificación de clase del tipo SET[ELEM], correspondiente al ejemplo de especificación algebraica presentado en la Sección 2.2.

En la cláusula `class-spec` es indicado el nombre del módulo de especificación de clase y los tipos genéricos son especificados en la cláusula `generic`. El tipo de interés es indicado en la cláusula `type of interest`, mientras que en la cláusula `types` se indica los nombres de módulos de tipos auxiliares utilizados en las operaciones del tipo de interés. La cláusula `inherits` indica el tipo de representación utilizado para modelar el tipo de interés y el nombre del módulo de especificación de clase asociado al tipo de representación es indicado en la cláusula `ref-spec`. La cláusula `operations` agrupa las definiciones de cada operación las cuales están conformadas por un encabezado (idéntico al utilizado en la interfaz de tipo) y por los términos de la pre y post-condición expresados en forma funcional.

```
class-spec: SET;
   generic: ELEM;
   type of interest: SET;
   types: NAT, BOOL;
   inherits: LIST;
   ref-spec: SET[ELEM];
   operations:
      ...
      DELETE(mod S:SET; E:ELEM);
         pre: IS_IN(S,E) is-true;
         post: S'=<REMOVE(S,E)>;
      ...
   end SET.
```

Fig. 3.- Especificación de Clase del tipo SET[ELEM]

2.4 Relaciones entre los objetos

En la Figura 4, es presentado un diagrama de clases en la notación UML que muestra las relaciones entre los objetos que intervienen en el modelo de desarrollo de software.

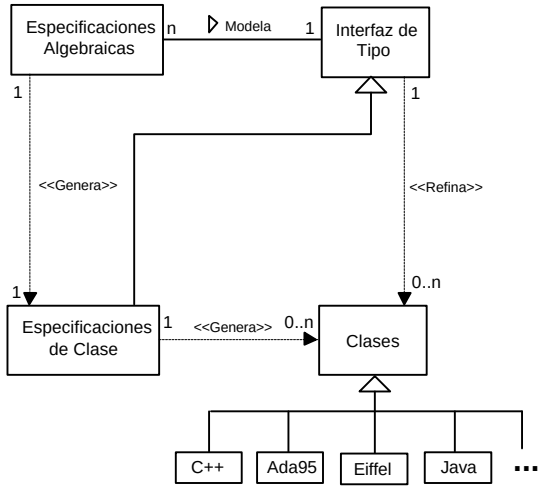

Fig. 4.- Diagrama de Clases UML de los objetos que conforman el modelo de desarrollo de software

En cuanto a las relaciones entre los objetos, se puede citar:

- Existe una relación de modelación entre una interfaz de tipo y múltiples especificaciones algebraicas, en el sentido de que un tipo cualquiera puede ser *modelado* por diversas especificaciones, dependiendo de la abstracción utilizada como base para la implementación.
- Una especificación de clase hereda de una interfaz de tipo la información de los encabezados y los tipos auxiliares. Por otra parte, una especificación algebraica permite *generar* las pre y post-condiciones de una especificación de clase, aplicando un proceso de transformaciones sucesivas sobre los axiomas de la especificación algebraica.
- Aplicando un proceso de derivación de código a partir de especificaciones de clase, se *genera* una clase concreta que implementa o *refina* las operaciones indicadas en una interfaz de tipo dada.
- Las clases pueden expresarse en diversos lenguajes de programación orientados a objeto, entre los cuales se pueden citar: C++, Java, Eiffel y Ada 95.

En las secciones 3 y 4 se presentan las transformaciones que conforman el modelo aplicadas sobre los diversos objetos descritos en esta sección.

3 TRANSFORMACIONES PARA GENERAR UNA ESPECIFICACION DE CLASE

Como se indicó al inicio de este artículo, el objetivo es presentar un conjunto de transformaciones que conforman un modelo de desarrollo de software. El proceso general de transformaciones puede ser denotado por:

$$ALG_{spec} \rightsquigarrow \cdots \rightsquigarrow CLASS_{spec} \rightsquigarrow \cdots \rightsquigarrow CLASS$$

donde, ALG_{spec} constituye una especificación algebraica de una abstracción de datos en términos de una abstracción de representación, $CLASS_{spec}$ es una especificación de clase (basada en pre y post-condiciones) generada a partir de ALG_{spec} y CLASS es la implementación (clase) de la abstracción en términos de un lenguaje de programación orientado a objeto.

En esta sección se presenta el primer bloque de transformaciones aplicadas a los axiomas de una especificación algebraica ALG_{spec}. El producto o salida de estas transformaciones es una especificación de clase $CLASS_{spec}$, equivalente a la especificación algebraica original. La especificación de clase introduce la noción de estado no presente en los axiomas de la especificación algebraica y además constituye la entrada principal de la transformación que permite generar el código o implementación de la abstracción en términos de una clase, proceso que será descrito en la Sección 4.

El proceso de transformaciones para generar especificaciones de clase está compuesto por dos transformaciones básicas: la transformación de los axiomas de la especificación de referencia del estilo constructivo al estilo funcional (TR_1) y la transformación a partir del estilo funcional de los axiomas a la especificación de clase (TR_2). Gráficamente, se tienen los siguientes pasos de transformación:

$$ALG_{spec} \rightsquigarrow_{TR1} [ALG_{spec}]^f \rightsquigarrow_{TR2} CLASS_{spec}$$

donde $[ALG_{spec}]^f$ representa la especificación obtenida con sus axiomas transformados al estilo funcional.

3.1 TR_1: Transformación de los axiomas de la Especificación Algebraica del estilo constructivo al estilo funcional

Como se indicó en la Sección 2.2, el modelo de desarrollo de software presentado utiliza como entrada especificaciones algebraicas de implementación abstracta, cuyos axiomas están expresados en el estilo *constructivo*; esto es, el conjunto de axiomas de las operaciones no constructoras están definidos de una manera completa en función de las constructoras.

La idea general de esta transformación es expresar los axiomas de las operaciones en base a los argumentos que corresponden al perfil de la operación; luego, cada axioma sería de la forma: $f(x_1,...,x_n) = t$, donde f representa una operación de la especificación y t es un término donde las únicas variables utilizadas son $x_1,...,x_n$.

Los pasos de esta transformación dependen de la forma de los axiomas de cada operación; el caso más general lo constituyen aquellas operaciones que poseen una definición completa ecuacional y en consecuencia sus axiomas en el estilo constructivo son de la forma:

$$f(c_1(...)) = t_1$$
$$\vdots$$
$$f(c_n(...)) = t_n$$

donde, f se asume unaria por simplicidad, $c_1,...,c_n$ son las operaciones constructoras de la especificación y cada término t_i ($i \in 1..n$) es un término con variables. Esta definición corresponde a una operación total, pero pueden existir operaciones no definidas para un constructor (operaciones parciales).

El proceso general de transformaciones para el conjunto de axiomas de una operación f (denotado por Ax_f) puede denotarse por:

$$Ax_f \rightsquigarrow_{cf1} [Ax_f]^c \rightsquigarrow_{cf2} [Ax_f]^v \rightsquigarrow_{cf3} [Ax_f]^p \rightsquigarrow_{cf4} [Ax_f]^f$$

donde, las transformaciones $cf_1,...,cf_4$ están definidas por:

- cf_1: $\forall A \in Ax_f$, se tiene que: $A = [f(c_i(...)) = t_i] \rightsquigarrow_{cf1} [x = c_i(...) \Rightarrow f(x) = t_i]$, siendo x una nueva variable (que no aparece en el axioma A original) de sort compatible con el constructor c_i. Como resultado de este paso, se obtiene el conjunto de axiomas equivalentes a Ax_f donde los constructores son sustituidos por una variable, lo cual es denotado por $[Ax_f]^c$.

- cf_2: $\forall A \in [Ax_f]^c$, se tiene que: $A = [x = c_i(...) \Rightarrow f(x) = t_i] \rightsquigarrow_{cf2} [x = c_i(...) \Rightarrow f(x) = t_{si}]$, donde, del término con variables t_{si} se han eliminado todas las variables distintas a las introducidas en la transformación cf_1, sustituyéndolas por los correspondientes selectores. El conjunto de axiomas resultante de eliminar estas variables es denotado $[Ax_f]^v$.

- cf_3: $\forall A \in [Ax_f]^v$, se tiene que: $A = [x = c_i(...) \Rightarrow f(x) = t_{si}] \rightsquigarrow_{cf3} [Pc_i(x) = true \Rightarrow f(x) = t_{si}]$, donde Pc_i constituye el predicado asociado al constructor $c_i(...)$. El conjunto de axiomas condicionales obtenido mediante esta transformación es denotado $[Ax_f]^p$.

- cf_4: Los axiomas condicionales pertenecientes al conjunto $[Ax_f]^p$ son fusionados en un único axioma para la operación f utilizando la operación if_then_else, quedando:

$$
\begin{aligned}
f(x) = \ & \text{if } Pc_1(x) \\
& \text{then } t_{S_1} \\
& \text{else } \text{if } Pc_2(x) \\
& \qquad \text{then } t_{S_2} \\
& \qquad \ldots \\
& \qquad \text{else } t_{S_n}
\end{aligned}
$$

Si la operación f es una operación parcial existe al menos un constructor $c_i \in \{c_1,...,c_n\}$ para el cual la operación no está definida; luego, para el conjunto de axiomas de la operación se aplican las transformaciones $cf_1,...,cf_4$ descritas anteriormente y para cada precondición de la forma f(x) is-defined iff $x \neq c_i(...)$ se aplica la transformación cf_1 obteniendo el axioma: f(x) is-defined iff $Pc_1(...)$ = false.

3.2 TR_2: Transformación de los axiomas en el estilo funcional en la Especificación de Clase

El objetivo de la transformación TR_2 consiste en ensamblar las operaciones de la especificación de clase $CLASS_{spec}$, donde cada operación es descrita mediante una precondición y una postcondición en el estilo funcional.

TR_2 puede ser vista como una función de ensamblaje del encabezado junto con la pre y post-condición, utilizando como entrada la especificación $[ALG_{spec}]^f$ obtenida mediante TR_1. Esto es:

$$TR_2 = build(fcl_1([ALG_{spec}]^f), fcl_2([ALG_{spec}]^f))$$

La función fcl_1 permite obtener el encabezado de la operación, mientras que fcl_2 obtiene la pre y post-condición resultante de aplicar la transformación TR_1.

A continuación es presentado un ejemplo de la aplicación de las transformaciones TR_1 y TR_2 sobre los axiomas en el estilo constructivo de una operación `height`, asumiendo que la operación `pop` es el selector asociado al constructor `push`:

Axiomas en el estilo constructivo	Axiomas en el estilo funcional	Especificación de Clase

`height(empty) = zero;`
`height(push(x,s)) =`
`  suc(height(s))`
	`height(z) = if is_empty(z)`
	`  then zero`
	`  else`
	`  suc(height(pop(z)))`
		HEIGHT(Z:STACK) return NAT;
		`post: result=if is_empty(z) then`
		`  zero`
		`  else`
		`  suc(height(pop(z)))`

4 TRANSFORMACIONES PARA GENERAR UNA CLASE

En esta sección son presentadas las transformaciones que completan el proceso definido por el modelo de desarrollo de software presentado. Estas transformaciones permiten obtener el código de una clase en un lenguaje Orientado a Objeto (OO) a partir de su especificación de clase y combinadas con las presentadas en la Sección 3 completan la secuencia de pasos sistemáticos para obtener el código de una clase, partiendo de la especificación algebraica de la abstracción.

El enfoque de implementación abstracta utilizado para las especificaciones algebraicas, permite describir el comportamiento abstracto de un tipo de interés T en términos de un tipo base TB, especificado e implementado. Esta relación se refleja en el módulo de especificación del sort t (correspondiente al tipo T), indicando que su especificación de representación corresponde al módulo de especificación de tb (indicado por la cláusula **`use:`** `TB;`). Una vez que se obtiene la especificación de clase del tipo T mediante las transformaciones presentadas en la Sección 3, la relación de representación es expresada mediante una cláusula de herencia (**`inherits:`** `TB;`), la cual indica el supertipo asociado al tipo T.

Desde el punto de vista del paradigma OO, se establece una relación de herencia simple entre el tipo de interés (implementado mediante una clase) y su tipo de representación. Las transformaciones presentadas en esta sección utilizan esta noción y establecen la relación de herencia simple según los mecanismos disponibles en cada uno de los lenguajes de programación utilizados (Ada 95, C++, Eiffel o Java).

El proceso que permite generar una clase está dividido en dos transformaciones: transformación de la especificación de clase a una clase independiente del lenguaje destino o *pseudo-clase* (TR_3) y la transformación de la pseudo-clase a una clase concreta (TR_4). Luego, los pasos de transformación del proceso pueden completarse con:

$$CLASS_{spec} \rightsquigarrow_{TR3} [CLASS]^{p\text{-}c} \rightsquigarrow_{TR4} CLASS$$

donde $[CLASS]^{p\text{-}c}$ representa la pseudo-clase asociada a la especificación de clase $CLASS_{spec}$.

4.1 TR_3: Transformación de la Especificación de Clase a una Pseudo-clase

Esta transformación tiene como objetivo fundamental obtener una pseudo-clase $[CLASS]^{p\text{-}c}$ a partir de una especificación de clase $CLASS_{spec}$. La especificación de clase $CLASS_{spec}$ es obtenida mediante la aplicación de las transformaciones TR_1 y TR_2 presentadas en la Sección 3. La pseudo-clase es una clase independiente del lenguaje destino y se introduce con la finalidad de establecer un proceso de generación de clases genérico, de manera que éste pueda ser extendido y/o adaptado a nuevos lenguajes de programación destino.

La transformación TR_3 es aplicada a cada operación de la especificación de clase, permitiendo transformar las pre y post-condiciones de cada operación en un código imperativo equivalente. Luego, este código es transformado mediante la transformación TR_4 a la sintaxis del lenguaje destino seleccionado. Básicamente, la transformación TR_3 convierte los términos de la pre y post-condición de estilo funcional a estilo imperativo, tomando en cuenta la información de cómo son implementadas las operaciones involucradas en los términos. Esta información, es tomada de la interfaz de tipo asociada a todos los tipos auxiliares que son importados en la especificación de clase (indicados por las cláusulas `type of interest`, `types` e `inherits`). Luego, TR_3 puede expresarse como:

$$TR_3(CLASS_{spec}, TYPE_{int}) = [CLASS]^{p\text{-}c}$$

donde $TYPE_{int}$ es el conjunto de interfaces de tipo importadas en la especificación de clase $CLASS_{spec}$.

La transformación TR_3 también preserva la información de control (encabezados) de la especificación de clase, tales como: el nombre del módulo, el nombre de la pseudo-clase, los componentes genéricos, los tipos/clases importados y el supertipo.

Las construcciones necesarias para representar las pseudo-clases generadas por TR_3 son: asignación, instrucción condicional (`if-then-else`), llamadas a procedimientos y/o funciones (incluyendo recursión), declaración de variables locales. Los encabezados de las operaciones son heredados directamente de la especificación de clase. Adicionalmente, se destaca la ausencia de variables globales.

La transformación TR_3 tiene como entrada la especificación de clase de una operación OP (encabezado, precondición y postcondición) y el conjunto $TYPE_{int}$ de interfaces de tipo. Luego, si tp y t constituyen respectivamente los términos de la pre y post-condición de una operación OP y si TR_3^t representa la transformación TR_3 a nivel de términos funcionales de la pre o post-condición se tiene que el código imperativo de OP es de la forma (esquema *a priori*):

$$\text{if } TR_3^t(tp) \text{ then } TR_3^t(t) \text{ else OP\_ERROR}$$

donde, el efecto del código asociado a la postcondición es producido si y sólo si la precondición se cumple. Adicionalmente, se establece un esquema defensivo donde para cada operación OP se dispone de una operación especial OP_ERROR, que es invocada en el caso que la precondición de OP no se satisfaga. Este esquema defensivo luego puede ser implementando mediante el manejo de excepciones provisto en los lenguajes destino.

Como fue indicado en la Sección 2.4, el encabezado de una operación de la especificación de clase puede corresponder al perfil de una operación modificadora (similares a procedimientos) o una función. En el caso de operaciones modificadoras se restringe a un único parámetro por referencia de la operación (denominado *parámetro* `mod`). Dependiendo del tipo de la operación OP a generar, el código $TR_3^t(t)$ correspondiente a la traducción de la postcondición debe completarse con una instrucción de retorno (si OP es una función) o una asignación sobre el parámetro `mod` (si OP es modificadora).

La transformación TR_3^t opera sobre dos tipos de términos: términos simples (aplicación funcional de operaciones) y términos condicionales (basados en la operación `if_then_else`). Es importante señalar que los términos asociados a las precondiciones son siempre términos simples, mientras que los términos de la postcondición pueden ser términos simples o condicionales.

Otro aspecto importante de la transformación TR_3^t, es que ésta puede agregar nuevas varibles al código imperativo generado, de manera que se corresponda con la especificación de clase inicial. Los detalles asociados al proceso de creación de nuevas variables se encuentran en [9].

Transformación asociada a un término condicional

Cuando el término t a transformar por TR_3^t es de la forma: t = if t_{cond} then t_{then} else t_{else},se cumple que:

$$TR_3^t(t) = \text{if } TR_3^t(t_{cond}) \text{ then } TR_3^t(t_{then}) \text{ else } TR_3^t(t_{else})$$

El resultado de transformar un término condicional cualquiera consiste en transformar los subtérminos t_{cond}, t_{then} y t_{else} vía TR_3^t y luego ensamblar una instrucción `if_then_else` con los códigos obtenidos. Cabe destacar que el término t_{cond} es un término simple, mientras que los términos t_{then} y t_{else} pueden ser términos simples o condicionales. En el caso que existan términos condicionales anidados, se tratan primero los más internos.

A continuación se presenta un ejemplo de la aplicación de la transformación TR_3^t sobre un término condicional:

Término a transfomar (t)	Resultados Parciales	Resultado final ($TR_3^t(t)$)

`t =if eq(x,first(l)) then`
`  tail(l)`
`else`
`  add(first(l),remove(x,tail(l)))`
donde:
t_{cond} = `eq(x,first(l))`
t_{then} = `tail(l)`
t_{else} = `add(first(l),remove(x,tail(l)))`
	$TR_3^t(t_{cond})$= `EQ(X,FIRST(L))`
	$TR_3^t(t_{then})$= `TAIL(L)`
	$TR_3^t(t_{else})$= `L_1 := L;`
	`TAIL(L_1);`
	`REMOVE(X,L_1);`
	`ADD(FIRST(L),L_1);`
		`if EQ(X,FIRST(L)) then`
		`  TAIL(L);`
		`else`
		`  L_1 := L;`
		`  TAIL(L_1);`
		`  REMOVE(X,L_1);`
		`  ADD(FIRST(L),L_1);`

En el ejemplo, se introdujo una nueva variable en la traducción de la parte `else` para garantizar que el código derivado se correspondiera con el término funcional original. La información sobre las variables introducidas por TR_3^t es utilizada para luego en el código final de la operación, agregar la correspondiente declaración de variables locales.

Transformación asociada a un término simple

Cuando el término t a transformar por TR_3^t es de la forma: t = f(t_1,...,t_n), donde cada subtérmino t_i es un término simple, la transformación $TR_3^t(t)$ consiste en tratar los subtérminos t_i de izquierda a derecha y en caso que existan subtérminos anidados, se tratan primero los más internos. Al terminar de tratar los subtérminos t_i, se procede a tratar el término t como tal.

Dependiendo si la operación f del término t es implementada como función o como modificadora, se tienen dos transformaciones posibles para el término t:

- Caso F modificadora: $TR_3^t(t)$ = <$stmt_{t1}$>;...;<$stmt_{tn}$>;F(...);
- Caso F función: $TR_3^t(t)$ = <$stmt_{t1}$>;...;<$stmt_{tn}$>; <ret_act> F(...);

donde, <$stmt_{ti}$> representa una lista de instrucciones secuenciales que se agregan antes de la llamada de F como resultado de la transformación del término t_i. La etiqueta <ret_act> colocada antes de la llamada de la función F, debe sustituirse por la palabra clave `return` o por una asignación del resultado de la función a una variable.

En el código generado se introducen instrucciones secuenciales como consecuencia de agregar llamadas a operaciones implementadas como modificadoras, por instrucciones de asignación y por instrucciones de retorno (`return`, aplicado al final de las funciones).

4.2 TR_4: Transformación de una pseudo-clase a una clase en un lenguaje OO

La transformación TR_4 culmina la cadena de refinamientos o transformaciones que conforman el modelo de desarrollo de software. En particular, a partir de una pseudo-clase, la transformación TR_4 produce una clase en términos de un lenguaje de programación orientado a objeto.

La transformación TR_4 consiste en una familia de transformaciones T, tal que se dispone de una transformación para cada lenguaje destino. El modelo propuesto propone transformaciones para los lenguajes Ada 95, Eiffel, C++ y Java, pero se pueden agregar transformaciones para otros lenguajes. Luego, se tiene el siguiente conjunto:

$$\mathsf{T} = \{ TR_4^{Ada}, TR_4^{C++}, TR_4^{Eiffel}, TR_4^{Java} \}$$

La transformación TR_4 puede ser vista como una función:

$$TR_4([CLASS]^{p\text{-}c}, pl) = F(pl)([CLASS]^{p\text{-}c}) = CLASS$$

donde $pl \in \{Ada, C++, Eiffel, Java\}$ y $F(i) = TR_4^i$.

Las acciones realizadas por la transformación TR_4 son totalmente dependientes del lenguaje de programación ya que están orientadas a las características sintácticas y de tipos propios de cada uno. En general, las transformaciones TR_4^i realizan las siguientes acciones:

- Expresar el código de la pseudo-clase generada por TR_3 en base a la sintaxis del lenguaje de programación destino.
- Establecer la correspondencia entre los tipos predefinidos utilizados y sus equivalentes en cada lenguaje.
- Sustituir las operaciones de los tipos predefinidos por las expresiones equivalentes dependiendo del lenguaje destino. Para esto, es necesario definir el conjunto de operaciones disponibles por cada tipo predefinido y sus correspondencias en los lenguajes destino. Esta información se encuentra detallada en [9].

Dependiendo del lenguaje de programación destino las transformaciones TR_4^i deben realizar acciones específicas según las características y/o convenciones del lenguaje destino, entre ellas se pueden citar:

- *Instanciar clases/tipos genéricos*: Si el lenguaje destino es Java, antes de comenzar generar el código de la clase en la sintaxis de Java es necesario instanciar la lista de componentes genéricos, debido a que Java no permite definir clases genéricas.
- *Derivar cláusulas iniciales de la clase*: esta acción consiste en escribir el código correspondiente al encabezado de la clase (para el caso de C++, Eiffel y Java) o el paquete (en el caso de Ada 95). Esto incluye las sentencias de herencia del tipo base, importación de clases auxiliares y declaración de componentes genéricos (si existen).
- *Detectar llamadas a operaciones constructoras*: Si el lenguaje destino es C++ o Java, es necesario determinar el conjunto de operaciones constructoras, con la finalidad de sustituir el nombre original de los constructores por el nombre de la clase asociada; esto, debido a que tanto C++ como Java denotan los constructores mediante el nombre de la clase y son diferenciados entre sí por perfil/aridad. Adicionalmente, debe tomarse en cuenta que los constructores son tratados como funciones.
- *Detectar el objeto invocador de un método***:** Para el caso de C++, Java y Eiffel, al traducir el código se debe considerar que el objeto que realiza la llamada a un método cualquiera no forma parte de la lista de argumentos de la operación; en consecuencia, el objeto invocador constituye un *parámetro implícito* del método. Tanto en C++ como en Java el objeto invocador de un método cualquiera es accesado como un atributo de nombre `this`, mientras que en Eiffel es denotado `Current`. En ambos casos, este atributo es un apuntador al objeto que realiza la llamada al método. Este tratamiento es aplicado a todas las operaciones excepto los constructores.
- *Escribir las llamadas a métodos/operaciones según las convenciones del lenguaje*: en C++, Eiffel y Java las llamadas a los métodos son denotadas como: `<object><sep><op_id>(<args>)`, donde `<object>` es el objeto que realiza la llamada a la operación `<op_id>` con los parámetros actuales `<args>` y `<sep>` es el separador ‘`.`’ o ‘`->`’.
- *Tratamiento asociado a operaciones parciales*: El esquema general de código asociado a operaciones parciales producido por la transformación TR_3 es adaptado al estilo de manejo de excepciones que maneje el lenguaje.

Debido a que esta transformación es la que introduce los detalles del lenguaje de programación, la información de los tipos predefinidos es de alta importancia para establecer una correspondencia entre los tipos abstractos utilizados desde la especificación algebraica de partida y los tipos propios de cada lenguaje. El conjunto de tipos predefinidos utilizados en el modelo incluye aquellos tipos disponibles en la mayoría de los lenguajes de programación: tipos elementales (booleanos, cadenas de caracteres, enteros, reales, etc.) y tipos estructurados (arreglos, registros, etc.). Para los tipos estructurados se dispone una plantilla (*template*) general que es instanciada de acuerdo al tipo de los elementos que lo conforman.

A manera de ejemplo de la aplicación de la tranformación TR_4, se presentan parcialmente los módulos de especificación de clase, pseudo-clase y clase en Java para el tipo de interés `SET`. En particular, se presentan las cláusulas básicas de cada módulo y la operación `DELETE`. Es importante notar que como el tipo de interés es genérico, éste debe ser instanciado al momento de generar mediante TR_4 el código de la clase en Java; en el ejemplo, se puede notar que el componente genérico

ELEM se sustituye por el tipo predefinido INT, que corresponde al tipo int de Java. Los módulos citados son mostrados a continuación:

Especificación de Clase	Pseudo-Clase	Clase en Java

class-spec: SET;
generic: ELEM;
type of interest: SET;
types: NAT, BOOL;
inherits: LIST;
ref-spec: SET[ELEM];
operations:
...
DELETE(mod S:SET; E:ELEM);
pre: IS_IN(S,E) is-true;
post: S'=<REMOVE(S,E)>;
...
end SET.
	pseudo-class: SET_CLASS;
	generic: ELEM;
	class: SET;
	use: NAT_CLASS, BOOL_CLASS;
	parent class: LIST;
	methods:
	...
	DELETE(mod S:SET; E:ELEM);
	begin
	if IS_IN(S,E)
	then REMOVE(S,E);
	else DELETE_ERROR;
	end;
	...
	end SET_CLASS.
		import java.lang.*;
		class SET extends LIST {
		...
		public void DELETE(int E) {
		try {
		if !(this.IS_IN(E))
		this.REMOVE(E);
		else
		throw new Exception();
		}
		catch (Exception e) {
		system.out.println("...");
		}
		}
		...
		}

En el ejemplo también se puede observar que la relación de herencia entre el tipo de representación y el tipo de interés es expresada en Java mediante el mecanismo de herencia (extends) que éste provee. Además de la traducción del código de la pseudo-clase a instrucciones en Java, la transformación TR_4 realizó el tratamiento para el parámetro implícito del método DELETE utilizando el objeto this en lugar de la variable mod S y se reescribieron las llamadas de las operaciones según las convenciones de Java. Finalmente, se puede observar el uso del mecanismo de excepciones de Java (basado en bloques try y manejadores catch) para la programación defensiva.

5 CONCLUSIONES

Las transformaciones presentadas en la Sección 3 y 4 permiten aplicar un proceso sistemático, que permite obtener la implementación de un tipo de interés cualquiera realizando pasos sucesivos de transformación partiendo de su especificación algebraica, hasta obtener una clase o implementación concreta en Ada 95, C++, Eiffel o Java. Este conjunto de pasos constituye un modelo de desarrollo de software basado en transformaciones, resumido como sigue:

$$ALG_{spec} \rightsquigarrow_{TR1} [ALG_{spec}]^{f} \rightsquigarrow_{TR2} CLASS_{spec} \rightsquigarrow_{TR3} [CLASS]^{p\text{-}c} \rightsquigarrow_{TR4} CLASS$$

donde, las transformaciones TR_2, TR_3 y TR_4 requieren como entrada adicional la información de las interfaces de tipo, tanto del tipo de interés como de los tipos auxiliares utilizados desde la especificación original ALG_{spec}.

Este enfoque tiene como ventaja que el producto obtenido (clases) es correcto con respecto a la entrada del proceso (especificaciones formales) debido a que las transformaciones que forman parte del proceso se suponen correctas. El modelo propuesto incluye la reutilización de componentes de software, dado que toda abstracción de datos es especificada e implementada mediante otra ya disponible. Inclusive, las nuevas abstracciones generadas pueden ser utilizadas como base para definir otras más complejas; todo esto basado en la noción de subtipos y herencia.

Otra ventaja del modelo de desarrollo propuesto es la posibilidad de definir abstracciones genéricas, indispensable para definir componentes de software contenedoras de manera abstracta, sin importar el tipo/clase de sus elementos. El método toma en cuenta que no todos los lenguajes destino proveen características de genericidad (Java, por ejemplo) y en consecuencia, la tranformación TR_4 permite instanciar los componentes genéricos asociándole clases/tipos concretos.

Otro aspecto importante a señalar sobre el modelo de desarrollo presentado es la independencia de éste con respecto a los lenguajes de programación destino. Al definir la transformación de la pseudo-clase en una clase expresada en un lenguaje de programación (TR_4) como una familia de funciones, permite que esta familia sea extendida agregando nuevas definiciones TR_4^i orientadas hacia un lenguaje i cualquiera (orientado a objeto e imperativo).

Aunque las especificaciones de clase en este trabajo son derivadas de manera automática a partir de los axiomas de una especificación algebraica, el enfoque de especificación de clase puede utilizarse de manera independiente como alternativa para la especificación formal de clases.

El éxito de un enfoque de desarrollo de software con base formal generalmente está relacionado con las herramientas de software que lo soportan, de manera que se reduzcan las complejidades de los aspectos formales. El ambiente AEsCoP [1] cumple con el objetivo de automatizar las transformaciones aquí presentadas y adicionalmente, provee al usuario editores amigables mediante los cuales se pueden crear las entradas del proceso (especificaciones algebraicas, interfaces de tipo) de una manera asistida. A pesar de estas facilidades, el usuario aún debe tener conocimientos de la teoría de tipos algebraicos, pero los aspectos de edición son facilitados considerablemente.

En general, el enfoque propuesto se diferencia de otros enfoques como Larch [4] y VDM [5] en el sentido que los pasos que se aplican son sistemáticos y el código final que se obtiene (imperativo y orientado a objetos) proviene de transformaciones sucesivas realizadas sobre los axiomas de la especificación algebraica original.

Como posibles extensiones, resulta necesario complementar el enfoque de especificación de clase de manera que introduzca características no contempladas del paradigma OO, tales como: herencia múltiple, clases abstractas, etc.

REFERENCIAS

[1] ACOSTA A., SCALISE E., SORIANO A. *AEsCoP: an Environment of Specification and Construction of Programs*. Proceeding ISAS'98, Vol. 1, 4th International Conference on Informaton Systems analysis and Synthesis. Orlando, USA, 1998.

[2] BERNOT G. *Correctness Proof for Abstract Implementations*. Information and Computation. Vol. 80, N° 2, pages 121-151, Feb. 1989.

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

[4] GUTTAG J.V., HORNING J.J. *LARCH: Languages and Tools for formal specification*. Texts and Monographs in Computer Science. Springer-Verlarg, 1993.

[5] JONES C. *VDM: Une Méthode rigourese pour le développement du logiciel*. Serie Méthodologies du logiciel. Masson, 1993.

[6] RUMBAUGH J., BOOCH G., JACOBSON I. *UML Unified Modeling Language*. Version 1.0, Rational Software Corporation, 1997. URL: http://www.rational.com

[7] SANNELLA D., TARLECKI A. *Esential concepts of algebraic specification and program development*. Department of Computer Science, University of Edinburgh, UK. Institute of Computer Science, Polish Academy of Sciences, Poland. August, 1996.

[8] SCALISE E. *SCAPrI: Sistema para la Construcción Automática de Programas Imperativos a partir de especificaciones formales*. Trabajo Especial de Grado. Centro ISYS. Escuela de Computación, Facultad de Ciencias, UCV, 1996.

[9] SCALISE E. *Un modelo de desarrollo de software para la generación automática de clases*. Trabajo Especial de Maestría. Postgrado en Ciencias de la Computación, Facultad de Ciencias, Universidad Central de Venezuela, 1999.

[10] SCALISE E., SORIANO A., ZAMBRANO N. *Derivación de código de clases a partir de especificaciones formales*. Panel'97. XXIII Conferencia Latinoamericana de Informática. Volumen 2, págs. 769-779. Valparaiso, Chile, 1997.

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