A Software Development Model for the Automatic Generation of Classes

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

ABSTRACT

In this paper it is presented a software development model based on transformations that allows to derive, in an automatic way, classes in object-oriented programming languages (Ada 95, C++, Eiffel and Java) starting from formal specifications. The set of transformations that conforms the software development model are a systematic steps, which starts from the algebraic specification of a type. This algebraic specification describes the abstract behavior of a type (type of interest) by means of other type, previously specified and implemented (representation type). In a progressive way, the transformations steps allow get a program (class) nearby to the initial specification (type of interest). These transformations obtain -in the first step- an intermediate specification (class specification) that it describes the operations of the type of interest by means of pre and post-conditions. Then, the intermediate specification is used to obtain imperative code in language-independent notation (pseudo-class); and finally the pseudo-class is transformed to any object-oriented programming language for which it has been defined transformations.

Keywords: software development model based on transformations, automatic construction of programs, formal specifications, algebraic specifications, class specifications, data type, class, object oriented languages.

1 Introduction

In this article is presented a software development model based on transformations whose main objective is to obtain the concrete implementation of a type (class) starting from the algebraic specification of the type by means of an abstract representation.

A software development model based on transformations has as main objective to build a program P that is equivalent to a specification SP (Mod(P) $\subseteq$ Mod(SP), where Mod(A) is the group of models that validates the sentences of A). The construction of the program P starting from the specification SP consists on a sequence of refinement steps:

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

In this chain, SP_0 is the original specification, SP_n is the program P and $SP_{i-1} \rightsquigarrow SP_i$ for i = 1,...,n is a refinement step. These successive steps guarantee that the derived program is equivalent to the original specification if the refinement steps have been proved; in other words, SP_{i-1} is refined as SP_i (denoted $SP_{i-1} \rightsquigarrow SP_i$) if all model of SP_i is a model of SP_{i-1}. These steps can be described in a systematic way by means of transformation rules that are correct if the conditions of the rules are satisfied. This avoids to prove the correctness from each transformation step to each problem, since it is enough to prove the correctness of the transformations and to guarantee in each step that the conditions are satisfied [10]. The transformations associated to each step can be described in an algorithmic or formal way, indicating the inputs, their transformations and the output. These ideas of successive transformations for the automatic production of code are the basis of the software development method that is presented in this article.

In this work we select an algebraic approach because it is suitable to the transformation model and it provides a formal base to prove that the class obtained by the transformations its correct with respect to the original specification. There are others approaches that uses the notions of executable specifications like rapid prototyping or even the automatic generation of functional programs starting from formal specifications. Another way to specify the system is using a object-oriented vocabulary by means of UML models (i.e. using Rational Rose, Enterprise Architect, etc.); and starting from these models, it can be generated the code of the classes.

The software development model presented here is an extension of a method for the automatic construction of imperative programs proposed in [15], which was optimized and implemented in [11]. The main extensions consist on obtaining a multi-language software development model with support of generic abstractions; also taking into account that the destination languages are object-oriented. The initial advances of this work were presented in [13]. The details of the software development model (as well as its theoretical justification) are presented in [12]. The work described in this paper

also has a formal definition of the semantics aspects of the specification formalism that was used; but it is not detailed in this paper.

This paper is structured as follows: Section 2 presents the objects that take part of the software development method as well as the relationships among them. Sections 3 and 4 presents the two blocks of transformation steps that conform the model. Finally, Section 5 presents the conclusions of this paper.

2 Model's Objects

In this section, the objects that are part of the software development method are described shortly. It is important to point out that the concepts of type and classes used in this work are based on the definitions presented in UML [9]. In this context, a type allows to specify the abstract objects and the behavior of the external operations, while the class is used to describe objects with concrete states and implementation of the operations (methods). According to these definitions, a type specifies a class and reciprocally, a class implements a type.

In the Figure 1, it is presented a UML class diagram that shows the relationships between the objects that take part of the software development model. The objects presented are: type interface, algebraic specificaction, class specification and class.

With respect to the relationships between the objects, we have:

- There is a modeling relationship between a type interface and multiple algebraic specifications. Indeed, any type can be *modeled* by several abstract implementation specifications, depending on the base type used for the implementation.
- A class specification *inherits* from a type interface the information of the operations headers and the auxiliary types. On the other hand, an algebraic specification allows to *generate* the pre and post-conditions of a class specification. To carry out this generation, it is applied a process of successive transformations on the axioms of the algebraic specification.
- Applying a process of automatic derivation of programs in object-oriented languages, a concrete class is *generated* starting from class specifications. This class implements or *refines* the operations indicated in a given type interface.
- The classes can be expressed in diverse object-oriented programming languages, such as: C++, Java, Eiffel and Ada 95.

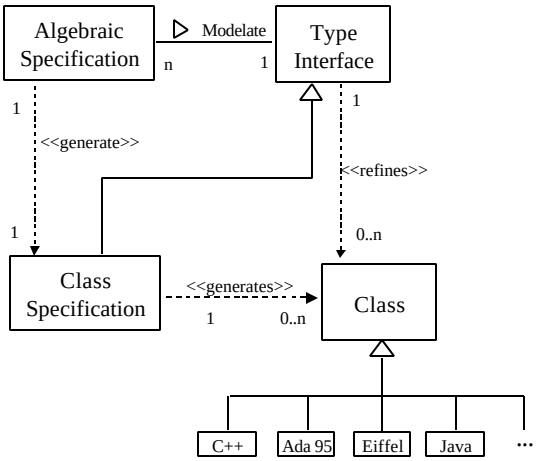

Fig. 1.- UML Class Diagram of the objects that conform the software development model

2.1 Type Interface

A type interface is the signature or public part of a data type; defined by a finite set of types (one type is distinguished as a special type, called the *type of interest*) and a set of operations with their respective profile. The profile of an operation is represented by means of a function or procedure header.

The type interface is the starting point to obtain the implementation of a type of interest, because this has the general aspects of the type of interest, such as: the operations of the type and the auxiliary types that are required. This information is useful for the users of the type and for the transformation processes, especially the process of automatic derivation of code, for which the type interface defines the "protocol" or the way that the operations should be implemented (functions or

procedures/modifiers). The interface type also defines the information about the formal parameters (order, type, parameter-passing methods, etc.).

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

Fig. 2.- Type Interface GEN_SET[ELEM]

As an example, in the Figure 2 is presented the interface of generic sets (`GEN_SET[ELEM]`). The clause `interface name` has the name of the type interface; it may contain optionally a list of generic components. The clause `type of interest` indicates the name of the type defined by the interface (type of interest). The clause `types` indicate the auxiliary types needed for the definition of the type of interest. Additionally, there is an optional clause `inherits`, used to create new types interfaces by means of the inheritance mechanism (specialization); by this way it is possible to enrich the super-type and even to redefine operations. In the clause `operations` are defined the header of the operations by means of selective operations (functions) and modifier operations (procedures). In the case of modifier operations, they can modify just one parameter and this is expressed by the keyword '`mod`'.

2.2 Algebraic Specification

An algebraic specification ALG_{spec} is a 3-tuple <S,Σ,Ax>, where S is the set of sorts, Σ is the set of operations and Ax is the set of axioms. The axioms of the specification express the abstract properties of the data type. In general, the axioms can be equations, conditional equations, formulas in first order logic, etc.

The algebraic specifications can be of two types:

– A *descriptive* algebraic specification that describes the abstract properties of the data type that is defined

– An *abstract implementation* specification, which describes the properties that characterize the implementation of the type of interest by means of another type that is supposed to be specified and implemented.

```
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. 3.- Algebraic Specification `SET(ELEM)`

The model presented in this paper is based on the abstract implementation approach, specifically [2]. Additionally, in this model we use a subset of PLUSS [3] as the specification language that permits to build complex, structured, modular and generic specifications.

There are some properties that the algebraic specifications used must accomplish:

– The axioms must be in constructive style

– There are operations that are inverse with respect to the constructors (called *selectors*)

– There are predicates to distinguish the values generated by the constructors.

– The axioms of a specification can be expressed as conditional equations.

In the Figure 3, it is presented the generic algebraic specification `SET(ELEM)`, which defines the abstract behavior of the set type (`SET`) by means of a generic list (`LIST`). The clause `generic spec` indicates the name of the algebraic specification module (generic in this case). The clause `use` list the default and auxiliary sorts imported by the module. The PLUSS language allows several clauses `use`, we add a second clause `use` to indicate which sort is the implementation sort. The clause `sort` is used to indicate the sort defined by the specification. The remaining clauses allows to define the profiles of the constructors (clause `generated by`) and not constructors (clause `operations`), the pre-conditions of the partial operations (clause `preconditions`), the axioms of the operations expressed as: equations, assertions or conditional equations (clause `axioms`) and the variables used in the axioms (clause `where`).

2.3 Class Specification

A class specification is a specification style where the operations of a class are specified by means of pre and post-conditions, based on a data abstraction. The precondition expresses the hypothesis related to the operations arguments and defines the conditions that have to be satisfied in order to apply the operation; the postcondition expresses the effect of the

operation. The class specification formalism is a part of the software development model presented in this paper; the details of this formalism as well as its semantics are presented in [12] and [14].

A class specification is defined by the tuple: $CLASS_{SPEC} = (<\sum_{interface}, ALG_{SPEC}, \mathcal{K}>, \wp)$, where: $\sum_{interface}$ is a *type interface*, ALG_{SPEC} is the reference algebraic specification, $\mathcal{K}$ is a mapping between the signatures $Sig(ALG_{SPEC})$ and $\sum_{interface}$. The pre and post-conditions of the operations in $\sum_{interface}$ are denoted by $\wp$.

The operations of a class specification are conformed by a *header* (inherited from the type interface), a precondition and a postcondition. The pre and post-conditions are functional terms with variables, derived from the axioms of the reference specification, by means of a transformation process presented in Section 3.

The operations of a class specification can be partial or total and depending on this, the precondition term can be optional. The postcondition term can have two possibilities depending if the operation is function or modifier. If the operation is a modifier, the post-condition is `post: x'=t`, where `x` is the reference parameter in the operation (`mod` parameter, in this model) and t is the value of the postcondition term. If the operation is a function, the postcondition is: `post: result=t`.

A class specification is the input of a process of automatic generation of code in diverse object-oriented languages (Ada 95, C++, Eiffel, Java, etc.), presented in Section 4.

```
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. 4.- Class Specification of the type `SET[ELEM]`

In the Figure 4, it is presented the class specification of the type `SET[ELEM]` whose algebraic specification is presented in Section 2.2.

Each class specification module has a name (clause `class-spec`). Additionally, if the class specification is derived of a generic algebraic specification, the generic components are indicated in the clause `generic`. The type of interest is indicated in the clause `type of interest`. The auxiliary types used in the operations of the type of interest are indicated in the clause `types`. The clause `inherits` indicates the super-type used to model the type of interest. The name of the class specification module associated to the representation type is indicated in the clause `ref-spec`. The clause `operations` contain the definitions of each operation, conformed by a header and the terms of the pre and post-condition (expressed as functional terms).

The transformations that are part of the software development model applied to the objects presented in this Section are presented in Section 3 and 4.

3 Class Specification Generation

As it was indicated to the beginning of this article, the aim of this paper is to present a set of transformations that conform a software development model. The general process of transformation can be denoted by:

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

where, ALG_{spec} is an algebraic specification of a data abstraction by means of a representation abstraction; $CLASS_{spec}$ is a class specification (with pre and post-conditions) obtained from ALG_{spec} and CLASS is the implementation (class) of the abstraction expressed in an object-oriented language.

In this section it is presented the first transformation steps applied to the axioms of the algebraic specification ALG_{spec}. The output of these transformations is a class specification $CLASS_{spec}$, equivalent to the original algebraic specification. The class specification has the notion of *state*, which is not present in the axioms of the algebraic specification; also it is the main input of the transformation that is used to generate the code or implementation by means of a class (see Section 4).

The transformation process applied to derive the class specifications is composed by two basic transformations: the transformation of the axioms of the reference algebraic specification from constructive style to functional style (TR_1) and the transformation from the axioms in functional style to the class specification (TR_2). This can be expressed by means of the following transformation steps:

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

where $[ALG_{spec}]^f$ is the original specification with axioms in functional style.

3.1 TR_1: Constructive Style to Functional Style

As it was indicated in the Section 2.2, the software development model uses abstract implementation specification as input, whose axioms are expressed in the constructive style; this is, the axioms of the non constructor operations are defined completely by means of the constructors.

The general idea of this transformation is to express the axioms of the operations by means of the operation arguments; then, each axiom will be defined by: $f(x_1,...,x_n) = t$, where f is an operation of the specification and t is a term where the only used variables are $x_1,...,x_n$.

The steps of this transformation depends on the kind of axioms from each operation; the general case is the operations with a complete equational definition and then its axioms in the constructive style are:

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

where, for simplicity f it is assumed an unary operation, $c_1,...,c_n$ are the constructors of the specification and each term t_i ($i \in 1..n$) is a term with variables. This is the definition of a total operation, but may exist operations that can't be defined to a constructor (partial operations).

The transformation process for the axioms of an operation f (denoted Ax_f) can be expressed as:

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

where, the transformations $cf_1,...,cf_4$ are:

- cf_1: $\forall A \in Ax_f : A = [f(c_i(...)) = t_i] \rightsquigarrow_{cf1} [x = c_i(...) \Rightarrow f(x) = t_i]$, where x is a new variable (that doesn't appear in the original axiom A) whose sort is compatible with the constructor c_i. As a result of this step, we obtain the set of axioms equivalent to Ax_f, where the constructors are substituted by a variable, which is denoted by $[Ax_f]^c$.

- cf_2: $\forall A \in [Ax_f]^c : A = [x = c_i(...) \Rightarrow f(x) = t_i] \rightsquigarrow_{cf2} [x = c_i(...) \Rightarrow f(x) = t_{si}]$, where, the term with variables t_{si} that are different than the variables introduced by cf_1, were substituted by the appropriate selectors. The set of axioms resultant is denoted $[Ax_f]^v$. The selector of each constructor is obtained by its profile; if there is no selector for a particular constructor, it must be added to the specification.

- cf_3: $\forall A \in [Ax_f]^v : A = [x = c_i(...) \Rightarrow f(x) = t_{si}] \rightsquigarrow_{cf3} [Pc_i(x) = true \Rightarrow f(x) = t_{si}]$, where Pc_i is the predicate for the constructor $c_i(...)$. The set of conditional axioms obtained by this transformation is denoted $[Ax_f]^p$.

- cf_4: The conditional axioms from the set $[Ax_f]^p$ are joined in one axiom for the operation f, using the if_then_else operation, then:

$$\begin{aligned} f(x) = \ & \text{if } Pc_1(x) \\ & \text{then } ts_1 \\ & \text{else if } Pc_2(x) \\ & \quad \text{then } ts_2 \\ & \quad \vdots \\ & \quad \text{else } ts_n \end{aligned}$$

If the operation f is a partial operation, there is at least one constructor $c_i \in \{c_1,...,c_n\}$ such that the operation is not defined; then, the set of axioms of f are transformed using $cf_1,...,cf_4$ and for each precondition: f(x) is-defined iff $x \neq c_i(...)$ it is applied the transformation cf_1 obtaining the axiom: f(x) is-defined iff $Pc_1(...) = false$.

3.2 TR_2: Functional Style to Class Specification

The transformation TR_2 assembles the operations of the class specification $CLASS_{spec}$, where each operation is defined by means of a pre and a post-condition expressed in functional style.

TR_2 can be interpreted as an assembling function of the header with the pre and post-condition, using as input the specification $[ALG_{spec}]^f$ obtained applying TR_1. This can be expressed as:

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

The function fcl_1 obtains the header of the operation and fcl_2 obtain the pre and post-condition result of applying the transformation TR_1. In this transformation it is used the mapping ϕ between the signatures $Sig(ALG_{SPEC})$ and $\Sigma_{interface}$. This

mapping makes a link between the operations in the algebraic specification and the operations of the type interface. This mapping is also defined to variables and sorts. In this work we use a syntactic convention to differentiate elements of $Sig(ALG_{SPEC})$ (lower-case letters) and $\Sigma_{interface}$ (upper-case letters).

Below is presented an example of applying the transformations TR_1 y TR_2 to the axioms in constructive style of an operation `height`. It is assumed that the operation `pop` is the selector for the constructor `push`:

Axioms in constructive style	Axioms in functional style	Class Specification

`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)))`

In this example, `pop` is the selector of the operation `push`; this is established by examining the operation's profile or by means of an user interaction[1]. Additionally, we can see the use of the mapping $\mathcal{L}$ for operations (`HEIGHT`) and sorts (`STACK`, `NAT`).

In [14] we use the notion of *institution* [5] to show the semantics of the class-specification and to establish the relation between the models that validates algebraic specifications and class-specifications models.

4 Class Generation

In this section the transformations needed to complete the software development model are described. These transformations obtain a class in an object-oriented language starting from its class specification; this transformation completes the chain of systematic steps to obtain a class starting from an algebraic specification of a data abstraction.

The abstract implementation approach allows to describe the abstract behavior of a type of interest T by means of a base type TB, which is specified and implemented. This relation is established in the specification module of the sort t (which defines the type T) indicating that the specification of the representation is the specification module of tb (syntactically, the clause **`use:`** `TB`). When the transformations presented in Section 3 are applied, in the class specification this relation is represented by means of an inheritance clause (**`inherits:`** `TB`), which indicates the supertype of T.

From the viewpoint of the object-oriented paradigm, it is established a simple inheritance relation between the type of interest (implemented by means of a class) and its representation type. The transformations presented in this section use this notion and they establish the single inheritance relation according to the features of the target programming languages (Ada 95, C++, Eiffel or Java).

The process used to generate a class is separated in two transformations: transformation from the class specification to a class in a language-independent notation or *pseudo-class* (TR_3) and the transformation from the pseudo-class to a concrete class (TR_4). Then, the transformation steps of this process are:

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

where $[CLASS]^{p\text{-}c}$ is the pseudo-class obtained from the class specification $CLASS_{spec}$ and $TYPE_{int}$ is the set of type interfaces imported in the class specification $CLASS_{spec}$.

4.1 TR_3: Class Specification to Pseudo-Class

The aim of this transformation is to obtain a pseudo-class $[CLASS]^{p\text{-}c}$ starting from the class specification $CLASS_{spec}$ that is obtained by means of the transformations TR_1 and TR_2 presented in Section 3. The pseudo-class is a class independent of the target language and it is introduced with the purpose of establish a generic process of class generation, so that this can be extended or adapted to other target programming languages.

The transformation TR_3 is applied to each operation of the class specification, allowing to transform the pre and post-conditions of each operation in an equivalent imperative code. Then, this code is transformed to the syntax of the target language by means of the transformation TR_4. Basically, the transformation translates the term of the pre and post-condition from functional style to imperative style, taking into account the information of how are implemented the operations used in the terms. This information is taken from the type interface of the auxiliary types imported from the class specification (clauses `type of interest`, `types` and `inherits`). Then, TR_3 can be expressed as:

[1] in a tool that supports this software development model the user interaction is useful in situations like this

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

where $TYPE_{int}$ is the set of type interfaces imported in the class specification $CLASS_{spec}$.

The transformation TR_3 also preserves the control information (headers) of the class specification, such as: the name of the module, the name of the pseudo-class, the generic components, the imported types/classes and the super-type.

The control structures needed to represent a pseudo-class generated by TR_3 are: assignment, conditional (`if-then-else`), procedure/function calls (including recursion) and local variables declaration. The headers of the operations are inherited directly from the class specification. Additionally, it is important to point out that there are not global variables.

The transformation TR_3 has as input the class specification of an operation OP (header, precondition and postcondition) and the set $TYPE_{int}$ of type interfaces. Then, if tp and t are the terms of the pre and post-condition of an operation OP and if TR_3^t is the transformation TR_3 applied to functional terms, the imperative code of OP is (*a priori* scheme):

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

where, the effect of the postcondition code is produced iff the precondition is satisfied. Additionally, it is established a defensive schema where for each operation OP we have a special operation OP_ERROR, which is called when the OP precondition is not satisfied. This defensive scheme can be implemented using the exception-handling mechanism provided by the target language.

As it was indicated in Section 2, the header of an operation from the class specification can correspond to a modifier operation (procedure) or a function. In the case of modifier operations there is only one reference parameter (`mod` parameter). Depending on the type of the operation OP, the translation of the postcondition ($TR_3^t(t)$) must be completed with a *return* statement (if OP is a function) or an assignment statement over the `mod` parameter (if OP is a modifier operation).

The transformation TR_3^t can be applied over two types of terms: simple terms (functional application of operations) and conditional terms (terms based on the `if_then_else` operation). It is important to emphasize that the precondition terms are always simple terms, while the postcondition terms can be simple or conditional terms.

Another important aspect of the transformation TR_3^t is that it can add new variables to the imperative code generated, such that it corresponds with the initial class specification. The details about this process are presented in [12].

Transformation of a conditional term

When the term to transform by TR_3^t is like: $t = \text{if } t_{cond} \text{ then } t_{then} \text{ else } t_{else}$, the output of TR_3^t is:

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

The result of transform any conditional term consists on transforms the sub-terms t_{cond}, t_{then} and t_{else} by means of TR_3^t and then assembles an `if_then_else` statement with the obtained codes. It is necessary to highlight that the term t_{cond} is a simple term, while the terms t_{then} and t_{else} can be simple or conditional terms. If there are conditional terms that are nested, the transformation is applied first to the inner terms.

Next, it is presented an example of the transformation TR_3^t applied to a conditional term:

Term to Transform (t)	Partial Results	Final Result($TR_3^t(t)$)

t = if eq(x,first(l)) then
tail(l)
else
add(first(l),remove(x,tail(l)))
where:
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);

In this example, it was added a new variable in the translation of the `else` part, to guarantee that the obtained code is equivalent to the original term. The information of the variables added by TR_3^t is used to add declaration statements in the final code of the operation.

Transformation of a simple term

When the term t to transform by TR_3^t is like: $t = f(t_1,...,t_n)$, where each subterm t_i is a simple term, the transformation $TR_3^t(t)$ must translate the subterms t_i from left to right and if there are nested sub-terms, it must translate de inner terms first. When all the terms t_i are translated, the transformation TR_3^t can translate the original term t. There are two possible transformations for the term t, depending if the operation f is implemented as a function or a modifier operation:

- Case F modifier operation: $TR_3^t(t) = <stmt_{t1}>;...;<stmt_{tn}>;F(...);$
- Case F function: $TR_3^t(t) = <stmt_{t1}>;...;<stmt_{tn}>; <ret\_act> F(...);$

where, $<stmt_{ti}>$ is a list of sequential statements that must be added before calling the operation F, as result of the translation of the term t_i. The label <ret_act> must be substituted by the keyword `return` or by an assignment of the result of the function to a variable.

In the generated code, there are new sequential statements as consequence of procedure calls, assignment statements and return statements (`return`, placed at the end of functions). All the transformations applied in this step are syntax-oriented.

4.2 TR_4: Pseudo-Class to Class Transformation

The transformation TR_4 ends the refinement steps or transformation that are part of the software development model. In particular, starting from a pseudo-class, the transformation TR_4 generates a class expressed in an object-oriented language.

The transformation TR_4 is a transformation family $\mathcal{T}$, such that there is a transformation to each target language. The proposed model has transformations for the languages: Ada 95, Eiffel, C++ and Java, but it can be added transformations for other languages. Then, the transformation family is:

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

The transformation TR_4 can be expressed as a function:

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

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

The actions performed by the transformation TR_4 are language-dependent because they are oriented to the syntax and types from each language. Generally, each transformation TR_4^i carries out the following actions:

- Write the pseudo-class code generated by means of TR_3, in the syntax of the target language.
- Establish a correspondence between the default types used in the pseudo-class and the equivalent type/class in each target language.
- Substitute the operations of the default types with the equivalent statements depending on the target programming language. To make this, it is necessary to define the set of operations for each default type and its correspondences in the target languages. This info is detailed in [12].

Depending on the target language each transformation TR_4^i must do specific actions according to the features or conventions of the target language, such as:

- *Instantiate generic classes/types*: If the target language is Java, it is necessary to instantiate the generic components list before to generate the class code in the Java syntax.
- *Generate the initial clauses of the class*: this action consists on write the code of the class header (in the case of C++, Eiffel and Java) or the specification package (in the case of Ada). This includes the inheritance of the base type, auxiliary class importation and generic component declaration (if they exist).
- *Detect calls to constructors*: If the target language is C++ or Java, it is necessary to determine the set of constructors, with the purpose of substitute the original name of the constructors with the name of the class; due to the fact that the constructors in C++ and Java have the same name as the class. Additionally, it must be considered that the constructors are like functions.
- *Detect the object that calls a method***:** In the case of C++, Java and Eiffel, when the code of the pseudo-class is translated, it must be considered that the object that makes the call to any method is not included in the arguments list of the operation. In consequence, the object that calls a method is an *implicit parameter* of the method. In Java and C++

the object that calls a method is a parameter whose name is `this`, while in Eiffel is called `Current`. In both cases, this attribute is a pointer to the object that calls the method. This detection is applied to all the operations except the constructors.

– *Write the calls to methods/operations in the syntax of the language*: in C++, Eiffel and Java the method calls are like: `<object><sep><op_id>(<args>)`, where `<object>` is the caller object of the method `<op_id>` with actual parameters `<args>` and `<sep>` is the separator '`.`' or '`->`'.

– *Treatment of partial operations*: The code of the partial operations generated with the transformation TR_3 is adapted to the exception-handling mechanism of the target language.

Due to the fact that this transformation introduce the details of the target language, the information of the default types is important to establish a correspondence between the abstract types (used from the original algebraic specification) and the types of each language. The sets of default types used in the model are: basic types (boolean, string, integer, float, etc.) and structured types (arrays, records, collections, etc.). A general template that is instantiated depending on the elements that compose it defines the structured types.

As an example of the application of TR_4, below are presented the class specification modules, pseudo-class and Java class for the type of interest `SET`. In particular, it is presented the basic clauses of each module and the operation `DELETE`. It is important to say that the type of interest must be instantiated at the moment of apply TR_4 to generate the Java code, because it is a generic type, and this feature is not supported by Java. In the example, the default type `INT` (which corresponds to the type `int` of Java) substitutes the generic component `ELEM`.

Class Specification

```
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

```
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.
```

Java 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("...");
    }
  }
  ...
}
```

In the example, it can be observed that the inheritance relationship between the representation type and the type of interest it is expressed in Java by means of the single inheritance mechanism (`extends`). The transformation TR_4 translated the code of the pseudo-class statements to Java, but also it wrote the method calls in the Java notation and it placed the `this` object instead of the `mod` parameter `S` in the `DELETE` method. Finally, it can be observed that the exception-handling mechanism of Java is used (`try-catch` blocks) to verify the preconditions.

5 Conclusions

The transformations presented in Sections 3 and 4 are used to apply a systematic process, which can be used to obtain the implementation of any type of interest carrying out successive transformation steps starting from the algebraic specification of the type, until obtain the class or concrete implementation in Ada 95, C++, Eiffel or Java. These steps conforms the software development model based on transformations, summarized as:

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

where, the transformations TR_2, TR_3 and TR_4 need as additional input the information of the type interfaces, from the type of interest and the auxiliary types used from the original specification ALG_{spec}.

This approach has as advantage that the class obtained is correct with respect to the input of the process (formal specification) because that the transformations that are part of the model are supposed to be correct. The proposed model includes the reuse of software components, due to every data abstraction is specified and implemented by means of other

available abstraction. Even, the new abstractions generated with the process can be used as base to define more complex abstractions, based on the sub-types and inheritance relationship.

Another advantage of the software development model is the possibility to define generic abstractions, which is useful to define abstract containers, whose elements are generic. Also, the model takes into account that the target language can't define generic abstraction, in such case, the transformation TR_4 allows to instantiate the generic components with concrete class/types.

Another important feature is that the software development model presented is programming-language independent. The definition of the transformation of the pseudo-class to a class in a programming language (TR_4) as a transformations family, allows that this family can be extended, adding a new transformation TR_4^i to any language i (object-oriented and imperative).

Although the class specifications used in this work are derived in an automatic way starting from the axioms of an algebraic specification, the class-specification formalism can be used independently, as an alternative way to specify classes formally.

The success of a software development model with formal base generally is related with the software tools that support it, as a way to decrease the complexities of the formal aspects. The AEsCoP environment [1] has as main objective to apply, in an automatic way, the transformations presented here and additionally, it provides friendly tools used to create the inputs of the process (algebraic specifications, type interfaces) in an assisted way. In spite of this, the user still needs to know the general aspects of algebraic types, but the edition is less complex.

In general, the proposed approach is different from other approaches such as Larch [4] and VDM [6] in the fact that the steps that are applied are systematic and the final code obtained (object-oriented and imperative) is derived from successive transformations over the axioms of the original algebraic specification.

As possible extensions of this work, it is necessary to supplement the class specification formalism with object-oriented features not contemplated here, such as: multiple inheritance, abstract classes, etc.

Finally, it is important to point that there are many works like this but in the context of functional languages or executable specifications; but in the field of procedural and object-oriented languages there are a little number of works (like [7], [8] and others).

References

[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 Information 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] GOGUEN J.A., BURSTALL R. *Institutions: Abstract Model Theory for Specification and Programming*. Journal of the ACM, Vol 39(1), pp. 95-146, January 1992.

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

[7] LIN H. *Procedural Implementation of Algebraic Specifications*. Institute of Software, Chinese Academy of Sciences, Beijing. ACM Transactions on Programming Languages and Systems, Vol. 15, No. 5, November 1993, pages 876-895.

[8] MIKHAJLOVA A., SEKERINSKI E. *Class Refinement and Interface Refinement in Object-Oriented Programs*. Proceedings of FME97, Graz, Austria, Lecture Notes in Computer Science, 1313, 1997.

[9] OMG UML Unified Modeling Language Specification. Version 1.3. June 1999. URL: http://www.omg.org/uml

[10] SANNELLA D., TARLECKI A. *Essential 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.

[11] 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.

[12] 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.

[13] 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.

[14] SCALISE E., ZAMBRANO N. *Semantics for Class Specification Formalism*. World Multiconference on Systemics, Cybernetics and Informatics, SCI'2000. Orlando, Florida, USA. Julio, 2000.

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