# Migration Revisited using the Categories Theory: Application to Biological Modelling Languages

Pierre Martin, Thérèse Libourel Rouge, Pascal Clouvel, Philippe Reitz

# Migration revisited using the Category Theory: Application to Biological Modelling Languages

Pierre Martin[1,2], Therese Libourel[2], Pascal Clouvel[3], and Philippe Reitz[2]

[1] CIRAD - PERSYST/DIR – Avenue Agropolis, 34398 Montpellier Cedex 5, France
Pierre.martin@cirad.fr
[2] LIRMM, CNRS and Université de Montpellier 2, 161, rue Ada,
34392 Montpellier cedex 5, France
{libourel, reitz}@lirmm.fr
[3] CIRAD – UPR 102 - Avenue Agropolis, 34398 Montpellier Cedex 5, France
Pascal.clouvel@cirad.fr

**Abstract.** Since the early seventies, numerous numerical programs have been developed to simulate biophysical processes. The current challenge facing scientific communities is to access any program using their usual language. Migration is a way of dealing with the challenge.
The conventional methodology to build up migration functions is the rule-based method. An alternative consists in using the category theory. This paper illustrates the methodology applied to bidirectional migration of a program written using a language based on the system theory (SIMILE) to a biological modelling language (APES - EU SEAMLESS integrated project) constructed using design patterns. Using the category theory enabled us (i) to provide a mathematical formalism for composition of the program features, i.e. the architecture and semantic function, (ii) to identify aspects which could not be preserved in the migration because of language expressiveness, and (iii) to automate the migration.

**Keywords:** Preservation, Architecture, Semantic Function, UML, Program Equivalence.

## 1 Introduction

Since the early seventies, numerous numerical programs have been developed to simulate biophysical processes. Most of those programs were first written using common programming languages, e.g. Fortran, and source code structure specific to each researcher [1]. Later on, in the 90s, a family of languages based on the system theory appeared [2]. Within that range, Modelica [3], Stella [4] and SIMILE [5], for instance, started being used as an assistant framework for program construction. Those languages used the concepts of object, relation between objects, object aggregation and model abstraction level M1. Recently, the new concept of design patterns [6] and ADL [7] started to be used in biological programs [8]. While languages based on the system theory and on the new concepts are simultaneously being used, the biological community today is faced with the problem of re-using the

different programs. The current challenge facing the community is to access any program using their usual language, and to be able to integrate program improvements whatever the language. Bidirectional migration [9, 10] is a way of dealing with the challenge.

To be moved at the end: Rule-based transformation systems [11] and triple graph grammars [12] are commonly adopted for migration. Both methods consist in identifying appropriate transformation rules and the rule matching algorithms [13]. In these methods, the transformation rules are incrementally identified and thus not automated. Moreover, preservation is evaluated *in fine*. Alternatively, other methodologies are quoted facing preservation. Concerning architecture, [14] proposed a method using process algebra. On the other hand, [15] adopted the category theory for preserving program maintainability in regard to the semantic function, i.e. the program computation [16]. Out of preservation, the category theory has already been applied to computer science to solve complex theoretical problems [17, 18].

Program = Turing machine, but need to preserve data organization for maintenance.

In the case of biological program migration, our objective was to preserve the architecture of the program and the semantic function. Our question was then to identify a methodology enabling bidirectional preservation of both features. The category theory [19] deals with mathematical structures and relationships between them. Graphs are made of objects, and arrows between objects. In addition to objects and arrows (called morphisms in the category theory), the category theory also considers the composition of arrows, i.e. the composition of morphisms [20]. The advantage of the theory is thus to take into account graph structure through the composition of morphisms. In our case, we assumed that languages could be represented by categories. We hypothesize that studying the relationship between categories could enable us to (i) identify aspects which could not be preserved because of the difference in language expressiveness and (ii) construct the appropriate bidirectional migration function. Once the object correspondence established in the scope of the category theory, the elaboration of translation rules becomes automated by construction.

The aim of this paper is to present the methodology and highlight its merits for bidirectional migration. We illustrate our proposal with the bidirectional migration of a program written using SIMILE [5] and a biological modelling language (APES - EU SEAMLESS integrated project) constructed using design patterns [6].

## 2 Category theory

### 2.1 Definition and notation

A category C is defined by:

- A class of objects, noted Ob(C).
- For each pair of objects (X, Y), a set noted $Hom_C(X, Y)$, whose elements are called morphisms of X on Y.

- and for each triplet of objects (X, Y, Z) of Ob(C), an application $Hom_C(X, Y) \times Hom_C(Y, Z) \longrightarrow Hom_C(X, Z)$ called composition of morphisms.

A category should verify the following conditions:

- Condition 1: if a pair of objects $(X_1, Y_1)$ is different from the pair $(X_2, Y_2)$, then
$$Hom_C(X_1, Y_1) \cap Hom_C(X_2, Y_2) = \varnothing$$
- Condition 2: for each object $X \in Ob(C)$, there is an element of $Hom_C(X, X)$, noted $id_X$ which is a neutral element for the composition of morphisms: $f \circ id_X = id_Y \circ f = f$
- Condition 3: composition is associative. Let $f: X \longrightarrow Y$, $g: Y \longrightarrow Z$ and $h: Z \longrightarrow T$, we have: $(h \circ g) \circ f = h \circ (g \circ f)$.

### 2.2 Operations on categories

Several operations can be performed on categories (monads, toposes, etc.). Two of them were used in this study: product of categories and functor. In our case, because the class of objects was a finite set and the class of morphisms was a set, the category is said to be small.

**Product of small categories.** Let $Ob(C) = \{X_C, Y_C \ldots\}$ and $Ob(D) = \{X_D, Y_D \ldots\}$. The product of C and D, noted $C \times D$, provides a new category E defined as follows:

- Ob(E) is the Cartesian product of Ob(C) and Ob(D). An object of E corresponds to a pair of objects of C and D, e.g. $(X_C, X_D)$.
- Hom(E) is the Cartesian product of Hom(C) and Hom(D). A morphism of E $(X_C, Y_C) \longrightarrow (X_D, Y_D)$ is a pair $<f, g>$ of morphisms of $C \times D$ where $f: X_C \longrightarrow X_D$ in C and $g: Y_C \longrightarrow Y_D$ in D.
- The composition of morphisms is defined by: $<f_C, g_C> \circ <f_D, g_D> = <f_C \circ f_D, g_C \circ g_D>$
- and morphism identity by: $id_E = <id_C, id_D>$

**Functor.** A functor corresponds to a morphism of categories. A functor supports the mapping from objects to objects, morphisms to morphisms, and preserves source, target, identities and composition. The covariant functor F (called functor in this paper) mapping category C to category D is defined as follows:

- Objects: for each $X \in Ob(C)$, $F(X) \in Ob(D)$
- Morphisms: for each $f \in Hom_C(X, Y)$, $F(f) \in Hom_D(F(X), F(Y))$

Two conditions have to be verified:

- Condition 4: relative to identity morphism $F(id_X) = id_{F(X)}$
- Condition 5: let f and g 2 morphisms of the source category: $F(g \circ f) = F(g) \circ F(f)$

## 3 Application to migration

### 3.1 Categories

A language is defined by its architecture and a semantic function. According to [5], architectural configurations, or topologies, are connected graphs of constituents and connectors that describe architectural structure. Architecture provides the inclusion relationships between constituents and sets of constituents. Architecture can then be represented by a category ($C_{Arch}$), where constituents are objects and morphisms the inclusion relationships between constituents. In that case, the composition of morphism is given by the association of morphisms e.g. ($A \in B$) and ($B \in C$) $\longrightarrow$ ($A \in C$). In accordance with this definition, conditions 1 and 3 are verified by construction. For condition 2, the identity morphisms correspond to the inclusion relationship mappings of an object with itself, which is true.
The semantic function deals with the numerical relationship existing between constituents. The semantic function can thus be considered as a calling sequence. It can then be represented by a category ($C_{SF}$) where constituents are objects and morphisms the calling sequence of the constituents. In that case, the composition of morphism is given by the order relationship e.g. ($A \leq B$) and ($B \leq C$) $\longrightarrow$ ($A \leq C$). In accordance with this definition, condition 1 is verified by construction. Since the composition of morphisms corresponds to the composite of sequence, which is also a sequence, condition 3 is also verified. Finally, the identity morphism for condition 2 corresponds to the empty path.

From a mathematical point of view, the product of categories allows to consider simultaneously the characteristics of both categories. In this paper, we deal with languages whose architecture and semantic function are independent. Thus, we considered the language as the product of the architecture and the semantic function according to equation (1).

$$C_{Lang} = C_{Arch} \times C_{SF} \,. \tag{1}$$

### 3.2 Migration function

Migration consists in establishing the correspondence between the source and the target language. A language is described using classes to instantiate in order to build the program. By definition, the functor represents the migration function. Object mapping corresponds to mapping the source language classes to the target ones, and morphism to the translation rules. Translation rules consider simultaneously both the architecture and semantic function. Functor F from source to target language, represented respectively by $C_{SourceLang}$ and $C_{TargetLang}$ categories, is noted:

$$F: C_{SourceLang} \longrightarrow C_{TargetLang} \,. \tag{2}$$

In order to introduce a common description of the language, we regroup the original language classes using UML. The extra classes created are then used for constructing the categories and the functors. By definition, a functor should always

map the source to target objects and morphisms. Aspects having no target correspondence cannot be preserved.

# 4 Illustration

## 4.1 SIMILE category ($C_S$)

SIMILE [5, 21] is a modelling language devoted to the simulation of differential dynamic systems [22]. In the user interface, SIMILE proposes a set of classes that need to be instantiated in order to build up the program. Change in entity status in relation to that of the connected ones is specified using numerical equations also written by users. In [5], the authors describe language functionalities, but not the language itself. From functionalities, we deduced the class diagram presented in Figure 1.

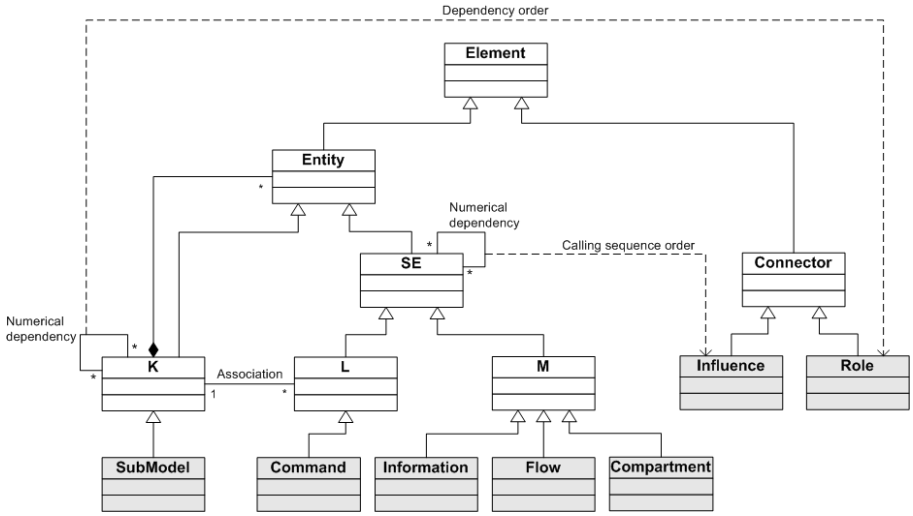


**Fig. 1.** Class diagram of SIMILE language designed using the UML convention. Classes are represented by rectangles. Classes in grey correspond to those being instantiated by the user. The diagram was inferred from the description of functionalities by [5].

In accordance with UML, the SIMILE language offers 2 types of element class: entity and connector, which correspond respectively to the objects used for the description of the simulated system, and to the characterization of the associative relationship existing between entities. We classified entities depending on their capacity to aggregate the others. Complex entities (K) could aggregate any entity while simple entities (SE) could not. Within the K class, "SubModel" enabled the constitution of groups of entities. Within the SE, we distinguished associative SE (L) which could be associated to K, and basic SE (M) which could not. M was the base class for 3 entity classes, namely:

- “Compartment”, representing the state of the system
- “Flow”, representing the numerical function responsible for changes in the state of the system
- “Information”, representing the information used in the management of the system. Information can be defined either by a parameter, or an input, or an intermediary variable.

L was the base class for the “Command” class associated to K. Commands were orders applied to every entity included in K (K entry condition, loops, etc.).

Two types of oriented connectors are offered. “Influence” is used to establish the numerical relationship between different SE and thus specify the calling sequence. “Role” only applies to K, and deals with the program architecture. It confers the size of the multi-dimensional array of a K target in relation to that of the K sources.

The class of objects of the two feature categories is a set containing the 3 objects of the language: {K, L, M}. Based on the class diagram (Figure 1), we established the morphisms between objects for the two feature categories shown in Figure 2. In the case of the semantic function, morphisms $K \longrightarrow M$ and $M \longrightarrow K$ are not explicit but resulted respectively from the composition of Hom(K, L) × Hom(L, M) and Hom(M, L) × Hom(L, K). From a mathematical point of view, the architecture feature graph presents a tree structure where leaves correspond to classes L and M, and internal nodes to class K. The semantic function feature graph presents a lattice structure where all classes are interconnected. These two graphs are summarised in Table 1.

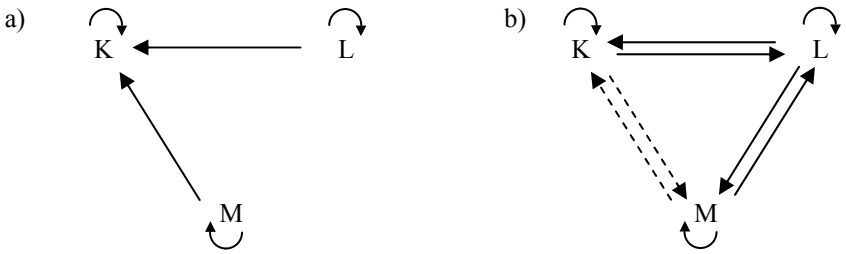


**Fig. 2.** Graphs of the architecture (a) and semantic function (b) of SIMILE.

**Table 1.** Morphisms of the architecture and the semantic function categories of SIMILE. A column corresponds to the source and a line to the target. $Id_X$ corresponds to the identity morphism, ‘*’ to the existence and ‘∅’ to the absence of morphism.

| Source object | Architecture | | | Semantic function | | |
|---|---|---|---|---|---|---|
| | K | M | L | K | M | L |
| K | $id_K$ | ∅ | ∅ | $id_K$ | * | * |
| M | * | $id_M$ | ∅ | * | $id_M$ | * |
| L | * | ∅ | $id_L$ | * | * | $id_L$ |

The language category corresponds to the product of the architecture and the semantic function categories. The resulting class of objects of the SIMILE category is $Ob(C_S)$ = {(K, K), (K, L), (K, M), (L, K), (L, L), (L, M), (M, K), (M, L), (M, M)}.

Language category morphisms are obtained by the Cartesian product of architecture and semantic function morphisms (Table 2).

**Table 2.** Morphisms of the SIMILE language category. The table shows the Cartesian product of morphisms of architecture (column), and semantic function categories (line). The table was drawn up from the definition of the product of categories according to the category theory. $Id_X$ corresponds to the identity morphism, '*' to the existence and '∅' to the absence of morphism.

| Hom(x,y) | K,K | K,M | K,L | M,K | M,M | M,L | L,K | L,M | L,L |
|---|---|---|---|---|---|---|---|---|---|
| K,K | $id_K$,$id_K$ | $id_K$,* | $Id_K$,* | ∅,$id_K$ | ∅,* | ∅,* | ∅,$id_K$ | ∅,* | ∅,* |
| K,M | $id_K$,* | $id_K$,$id_M$ | $Id_K$,* | ∅,* | ∅,$id_M$ | ∅,* | ∅,* | ∅,$id_M$ | ∅,* |
| K,L | $id_K$,* | $id_K$,* | $Id_K$,$id_L$ | ∅,* | ∅,* | ∅,$id_L$ | ∅,* | ∅,* | ∅,$id_L$ |
| M,K | *,$Id_K$ | *,* | *,* | $id_M$,$Id_K$ | $id_M$,* | $id_M$,* | ∅,$id_K$ | ∅,* | ∅,* |
| M,M | *,* | *,$id_M$ | *,* | $id_M$,* | $id_M$,$id_M$ | $id_M$,* | ∅,* | ∅,$id_M$ | ∅,* |
| M,L | *,* | *,* | *,$id_L$ | $id_M$,* | $id_M$,* | $id_M$,$id_L$ | ∅,* | ∅,* | ∅,$id_L$ |
| L,K | *,$id_K$ | *,* | *,* | ∅,$id_K$ | ∅,* | ∅,* | $id_L$,$id_K$ | $id_L$,* | $id_L$,* |
| L,M | *,* | *,$id_M$ | *,* | ∅,* | ∅,$id_M$ | ∅,* | $id_L$,* | $id_L$,$id_M$ | $id_L$,* |
| L,L | *,* | *,* | *,$id_L$ | ∅,* | ∅,* | ∅,$id_L$ | $id_L$,* | $id_L$,* | $id_L$,$id_L$ |

### 4.2 APES category ($C_A$)

APES [23] is a platform devoted to the simulation of agricultural production on a field level. APES was developed under the EU Sixth Framework Research Programme SEAMLESS [24, 25]. APES architecture consists of a set of self-contained components [26] interconnected using the ModCom core [27]. The role of Modcom is (i) to construct a recursive calling chain of the component set and (ii) to transfer output variable values from one component to the others at every time step of the simulation. In APES, the architecture of a component obeys a specific design based on the use of design patterns. The connection of the component with ModCom is based on the "adapter" structural design pattern. Connection consists in adapting the component interface (ClsMyComponent) to that of ModCom (OdeSimObj). OdeSimObj provides four virtual methods which present specific roles in the component running within the components chain. Among them, GetRates is the method dedicated to run the program of the component (Figure 3).

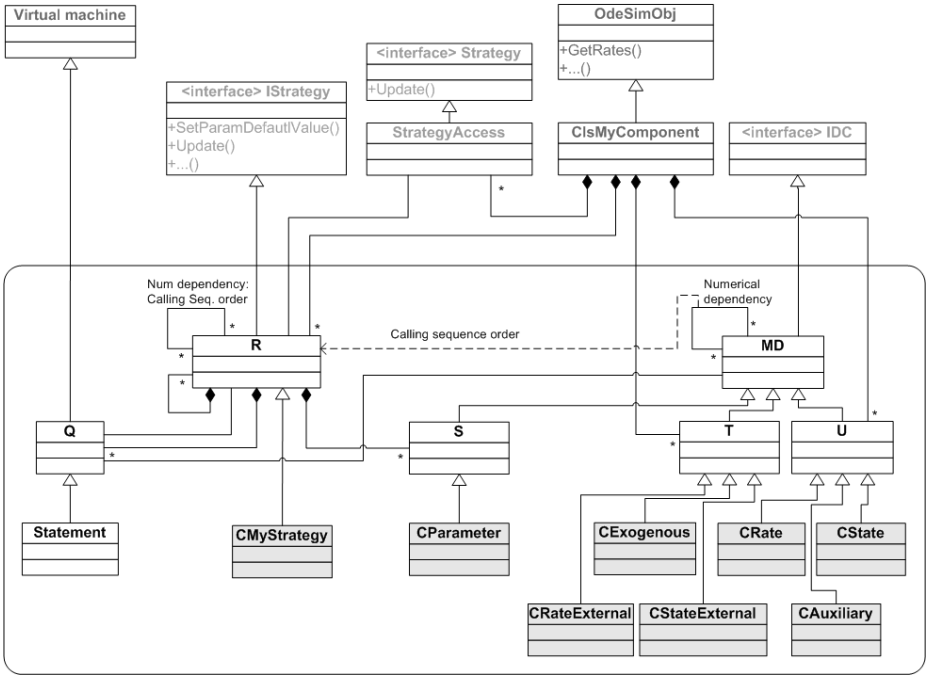


**Fig. 3.** Class diagram of an APES component designed using the UML convention. Classes in grey correspond to those implementing the component program.

The structure of the component program is based on the object paradigm [28]. In accordance with this paradigm, the state of the program is described using classes of data (Implemented data class = MD) and its behaviour using classes of methods (Implemented method class = R). We established the class diagram of the component in reference to [23] and [29]. The role of data classes is to group variables in accordance with the categorization imposed by APES. A distinction is made between three types of data classes. Basic data (U) correspond to the variables of the program:

- "CState", containing the state variables of the program
- "CRate", containing the variation rate variables of the program
- "CAuxiliary", containing the other variables of the program which cannot be part of the two data classes above.

Input data (T) are provided from the other components of the platform:

- "CRateExternal", containing the input state variables
- "CStateExternal", containing the input variation rate variables
- "CExogeneous", containing the input auxiliary variables

Specific input data (S) relative to a particular method:

- "CParameter", containing the value of the different parameters of the program

In the data classes, the variable types correspond to usual static data types (integer, Boolean, real, string). Arrays of variables are allowed, but not data structures. Classes allow the adjunction of text describing each data item, its initial value and the upper and lower bounds.

The role of the method class is to establish the calling sequence of the methods and to carry out the calculation on the U variables using variables from U, T, and S. The R class is designed using three design patterns [6]: the "strategy" behavioural design pattern consists in defining a unique design of method structure in order to make them interchangeable, the "façade" structural design pattern consists in defining a unified interface, and the "composite" structural design pattern consists in allowing the composite of methods into tree structures to represent part-whole hierarchies of methods. The association of those patterns provides a unique structure to all methods, independently of the program source code.

Finally, we added the "Q" class corresponding to the statement provided by the virtual machine. In APES, the considered statements are "if" and "for".

The class of objects of the two feature categories is a set containing the 5 objects of the language: {R, U, T, S, Q}. Based on the class diagram (Figure 3), we established the morphisms between objects for the two feature categories (Figure 4). The graph of architecture feature shows 3 sub-graphs. Two sub-graphs consist of isolated nodes, i.e. U and T. The third sub-graph consists of a tree graph where leaves correspond to S and Q and the internal node to R. The graph of the semantic function feature shows a lattice structure. These two graphs are summarized in Table 3.

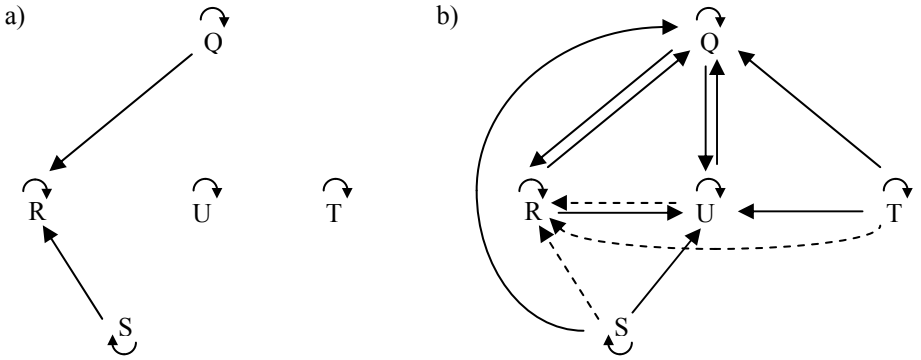


**Fig. 4.** Graphs of the architecture (a) and semantic function (b) of APES.

**Table 3.** Morphisms of the architecture and the semantic function categories of APES. A column corresponds to the source and a row to the target. $Id_X$ corresponds to the identity morphism, '*' to the existence and '∅' to the absence of morphism.

| Source | Architecture | | | | | Semantic function | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Object | R | U | T | S | Q | R | U | T | S | Q |
| R | $id_R$ | ∅ | ∅ | ∅ | ∅ | $id_R$ | * | ∅ | ∅ | * |
| U | ∅ | $id_U$ | ∅ | ∅ | ∅ | * | $id_U$ | ∅ | ∅ | * |
| T | ∅ | ∅ | $id_T$ | ∅ | ∅ | * | * | $id_T$ | ∅ | * |
| S | * | ∅ | ∅ | $id_S$ | ∅ | * | * | ∅ | $id_S$ | * |
| Q | * | ∅ | ∅ | ∅ | $Id_Q$ | * | * | ∅ | ∅ | $Id_Q$ |

The resulting class of objects of the APES category is $Ob(C_A)$ = {(R, R}, (R, U), (R, T), (R, S), (R, Q), (U, R), (U, U), (U, T), (U, S), (U, Q), (T, R), (T, U), (T, T), (T, S), (T,Q), (S, R), (S, U), (S, T), (S, S), (S, Q), (Q, R), (Q, U), (Q, T), (Q, S), (Q, Q)}. Language category morphisms are given in Table 4 (Q class not displayed).

**Table 4.** Morphisms of the APES language category. The table displays the Cartesian product of morphisms of architecture (column), and semantic function categories (row). The table is based on the definition of the product of categories according to the category theory. $Id_X$ corresponds to the identity morphism, '*' to the existence and '∅' to the absence of morphism. To improve legibility, '—' is used in place of '(∅,∅)'.

| Hom(x,y) | R,R | R,U | R,T | R,S | U,R | U,U | U,T | U,S | T,R | T,U | T,T | T,S | S,R | S,U | S,T | S,S |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R,R | $id_R$,$id_R$ | $id_R$,* | $id_R$,∅ | $id_R$,∅ | ∅,$id_R$ | ∅,* | — | — | ∅,$id_R$ | ∅,* | — | — | ∅,$id_R$ | ∅,* | — | — |
| R,U | $id_R$,* | $id_R$,$id_U$ | $id_R$,∅ | $id_R$,∅ | ∅,* | ∅,$id_U$ | — | — | ∅,* | ∅,$id_U$ | — | — | ∅,* | ∅,$id_U$ | — | — |
| R,T | $id_R$,* | $id_R$,* | $id_R$,$id_T$ | $id_R$,∅ | ∅,* | ∅,* | ∅,$id_T$ | — | ∅,* | ∅,* | ∅,$id_T$ | — | ∅,* | ∅,* | ∅,$id_T$ | — |
| R,S | $id_R$,* | $id_R$,* | $id_R$,∅ | $id_R$,$id_S$ | ∅,* | ∅,* | ∅,∅ | ∅,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ |
| U,R | ∅,$id_R$ | ∅,* | — | — | $id_U$,$id_R$ | $id_U$,* | $id_U$,∅ | $id_U$,∅ | ∅,$id_R$ | ∅,* | — | — | ∅,$id_R$ | ∅,* | — | — |
| U,U | ∅,* | ∅,$id_U$ | — | — | $id_U$,* | $id_U$,$id_U$ | $id_U$,∅ | $id_U$,∅ | ∅,* | ∅,$id_U$ | — | — | ∅,* | ∅,$id_U$ | — | — |
| U,T | ∅,* | ∅,* | ∅,$id_T$ | — | $id_U$,* | $id_U$,* | $id_U$,$id_T$ | $id_U$,∅ | ∅,* | ∅,* | ∅,$id_T$ | — | ∅,* | ∅,* | ∅,$id_T$ | — |
| U,S | ∅,* | ∅,* | — | ∅,$id_S$ | $id_U$,* | $id_U$,* | $id_U$,∅ | $id_U$,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ |
| T,R | ∅,$id_R$ | ∅,* | — | — | ∅,$id_R$ | ∅,* | — | — | $id_T$,$id_R$ | $id_T$,* | $id_T$,∅ | $id_T$,∅ | ∅,$id_R$ | ∅,* | — | — |
| T,U | ∅,* | ∅,$id_U$ | — | — | ∅,* | ∅,$id_U$ | — | — | $id_T$,* | $id_T$,$id_U$ | $id_T$,∅ | $id_T$,∅ | ∅,* | ∅,$id_U$ | — | — |
| T,T | ∅,* | ∅,* | ∅,$id_T$ | — | ∅,* | ∅,* | ∅,$id_T$ | — | $id_T$,* | $id_T$,* | $id_T$,$id_T$ | $id_T$,∅ | ∅,* | ∅,* | ∅,$id_T$ | — |
| T,S | ∅,* | ∅,* | — | ∅,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ | $id_T$,* | $id_T$,* | $id_T$,∅ | $id_T$,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ |
| S,R | *,$id_R$ | *,* | *,∅ | *,∅ | ∅,$id_R$ | ∅,* | — | — | ∅,$id_R$ | ∅,* | — | — | $id_S$,$id_R$ | $id_S$,* | $id_S$,∅ | $id_S$,∅ |
| S,U | *,* | *,$id_U$ | *,∅ | *,∅ | ∅,* | ∅,$id_U$ | — | — | ∅,* | ∅,$id_U$ | — | — | $id_S$,* | $id_S$,$id_U$ | $id_S$,∅ | $id_S$,∅ |
| S,T | *,* | *,* | *,$id_T$ | *,∅ | ∅,* | ∅,* | ∅,$id_T$ | — | ∅,* | ∅,* | ∅,$id_T$ | — | $id_S$,* | $id_S$,* | $id_S$,$id_T$ | $id_S$,∅ |
| S,S | *,* | *,* | *,∅ | *,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ | $id_S$,* | $id_S$,* | $id_S$,∅ | $id_S$,$id_S$ |

### 4.3 Migration

**From SIMILE to APES (F: $C_S \longrightarrow C_A$).** As architectural and semantic function categories are composed of the same set of objects, pair-to-pair correspondence can be summarized as object-to-object correspondence: K, L, and M in SIMILE correspond respectively to R, Q, and [U, T and S] in APES (Table 5).

**Table 5.** Summarized mapping of the language classes from SIMILE to APES.

| $Ob(C_S)$ | Simile class | | | APES class | $Ob(C_A)$ |
|---|---|---|---|---|---|
| K | Submodel | | | CMyStrategy | R |
| M | Compartment | | | CRate | U |
| M | Flow | | | CState | U |
| | | Parameter | | CParameter | S |
| | | Input | Connected to a compartment | CStateExternal | T |
| M | Information | | Connected to a flow | CRateExternal | T |
| | | | Other | CExogeneous | T |
| | | Intermediary variable | | CAuxiliary | U |
| L | Command | | | Statement | Q |

Establishing the correspondence of morphisms amounts to checking the cell content correspondence between Table 2 (source) and Table 4 (target). As they correspond to input data, S and T could not be the target of the semantic function morphisms (calling sequence). In Table 6, the target cells whose source content are not preserved in the migration are shown in grey. All grey cells correspond to architecture, i.e. (*,-) $\longrightarrow$ (∅,-). For Q, architecture and semantic function are preserved (not shown).

**Table 6.** Morphism preservation in the SIMILE to APES migration. In grey, cells whose source content is not preserved in the migration. All grey cells correspond to architecture, i.e. (*,-) $\longrightarrow$ (∅, -). To improve legibility, '–' is used in place of '(∅,∅)'.

| Hom(x,y) | R,R | R,U | R,T | R,S | U,R | U,U | U,T | U,S | T,R | T,U | T,T | T,S | S,R | S,U | S,T | S,S |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R,R | $id_R$,$id_R$ | $id_R$,* | $id_R$,∅ | $id_R$,∅ | ∅,$id_R$ | ∅,* | — | — | ∅,$id_R$ | ∅,* | — | — | ∅,$id_R$ | ∅,* | — | — |
| R,U | $id_R$,* | $id_R$,$id_U$ | $id_R$,∅ | $id_R$,∅ | ∅,* | ∅,$id_U$ | — | — | ∅,* | ∅,$id_U$ | — | — | ∅,* | ∅,$id_U$ | — | — |
| R,T | $id_R$,* | $id_R$,* | $id_R$,$id_T$ | $id_R$,∅ | ∅,* | ∅,* | ∅,$id_T$ | — | ∅,* | ∅,* | ∅,$id_T$ | — | ∅,* | ∅,* | ∅,$id_T$ | — |
| R,S | $id_R$,* | $id_R$,* | $id_R$,∅ | $id_R$,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ |
| U,R | ∅,$id_R$ | ∅,* | — | — | $id_U$,$id_R$ | $id_U$,* | $id_U$,∅ | $id_U$,∅ | ∅,$id_R$ | ∅,* | — | — | ∅,$id_R$ | ∅,* | — | — |
| U,U | ∅,* | ∅,$id_U$ | — | — | $id_U$,* | $id_U$,$id_U$ | $id_U$,∅ | $id_U$,∅ | ∅,* | ∅,$id_U$ | — | — | ∅,* | ∅,$id_U$ | — | — |
| U,T | ∅,* | ∅,* | ∅,$id_T$ | — | $id_U$,* | $id_U$,* | $id_U$,$id_T$ | $id_U$,∅ | ∅,* | ∅,* | ∅,$id_T$ | — | ∅,* | ∅,* | ∅,$id_T$ | — |
| U,S | ∅,* | ∅,* | — | ∅,$id_S$ | $id_U$,* | $id_U$,* | $id_U$,∅ | $id_U$,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ |
| T,R | ∅,$id_R$ | ∅,* | — | — | ∅,$id_R$ | ∅,* | — | — | $id_T$,$id_R$ | $id_T$,* | $id_T$,∅ | $id_T$,∅ | ∅,$id_R$ | ∅,* | — | — |
| T,U | ∅,* | ∅,$id_U$ | — | — | ∅,* | ∅,$id_U$ | — | — | $id_T$,* | $id_T$,$id_U$ | $id_T$,∅ | $id_T$,∅ | ∅,* | ∅,$id_U$ | — | — |
| T,T | ∅,* | ∅,* | ∅,$id_T$ | — | ∅,* | ∅,* | ∅,$id_T$ | — | $id_T$,* | $id_T$,* | $id_T$,$id_T$ | $id_T$,∅ | ∅,* | ∅,* | ∅,$id_T$ | — |
| T,S | ∅,* | ∅,* | — | ∅,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ | $id_T$,* | $id_T$,* | $id_T$,∅ | $id_T$,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ |
| S,R | *,$id_R$ | *,* | *,∅ | *,∅ | ∅,$id_R$ | ∅,* | — | — | ∅,$id_R$ | ∅,* | — | — | $id_S$,$id_R$ | $id_S$,* | $id_S$,∅ | $id_S$,∅ |
| S,U | *,* | *,$id_U$ | *,∅ | *,∅ | ∅,* | ∅,$id_U$ | — | — | ∅,* | ∅,$id_U$ | — | — | $id_S$,* | $id_S$,$id_U$ | $id_S$,∅ | $id_S$,∅ |
| S,T | *,* | *,* | *,$id_T$ | *,∅ | ∅,* | ∅,* | ∅,$id_T$ | — | ∅,* | ∅,* | ∅,$id_T$ | — | $id_S$,* | $id_S$,* | $id_S$,$id_T$ | $id_S$,∅ |
| S,S | *,* | *,* | *,∅ | *,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ | ∅,* | ∅,* | — | ∅,$id_S$ | $id_S$,* | $id_S$,* | $id_S$,∅ | $id_S$,$id_S$ |

**From APES to SIMILE (G: $C_A \longrightarrow C_S$).** Table 7 shows the mapping of the language objects from APES to SIMILE. The same methodology as applied to F enabled us to identify the preservation of G migration. In that direction all morphisms were mapped. All aspects of the 2 features are then preserved.

**Table 7.** Summarized mapping of the classes from APES to SIMILE.

| $Ob(C_A)$ | APES classes | SIMILE classes | $Ob(C_S)$ |
|---|---|---|---|
| R | CMyStrategy | Submodel | K |
| U | CState | Compartment | M |
| U | CRate | Flow | M |
| U | CAuxiliary | Information | M |
| T | CStateExternal | Information | M |
| T | CRateExternal | Information | M |
| T | CExogeneous | Information | M |
| S | CParameter | Information | M |
| Q | Statement | Command | L |

Verification of identity morphism mapping (condition 4) consisted in checking that the identity morphisms of APES corresponds to the identity morphisms of SIMILE. Table 8 shows that correspondence.

**Table 8.** Identity morphism correspondence between APES and SIMILE.

| $Ob(C_A)$ | $id_{APES}$ | $Ob(C_S)$ | $id_{Sim}$ | $Ob(C_A)$ | $id_{APES}$ | $Ob(C_S)$ | $id_{Sim}$ |
|---|---|---|---|---|---|---|---|
| R,R | $id_R$,$id_R$ | K,K | $id_K$,$id_K$ | T,R | $id_T$,$id_R$ | M,K | $id_M$,$id_K$ |
| R,U | $id_R$,$id_U$ | K,M | $id_K$,$id_M$ | T,U | $id_T$,$id_U$ | M,M | $id_M$,$id_M$ |
| R,T | $id_R$,$id_T$ | K,M | $id_K$,$id_M$ | T,T | $id_T$,$id_T$ | M,M | $id_M$,$id_M$ |
| R,S | $id_R$,$id_S$ | K,M | $id_K$,$id_M$ | T,S | $id_T$,$id_S$ | M,M | $id_M$,$id_M$ |
| U,R | $id_U$,$id_R$ | M,K | $id_M$,$id_K$ | S,R | $id_S$,$id_R$ | M,K | $id_M$,$id_K$ |
| U,U | $id_U$,$id_U$ | M,M | $id_M$,$id_M$ | S,U | $id_S$,$id_U$ | M,M | $id_M$,$id_M$ |
| U,T | $id_U$,$id_T$ | M,M | $id_M$,$id_M$ | S,T | $id_S$,$id_T$ | M,M | $id_M$,$id_M$ |
| U,S | $id_U$,$id_S$ | M,M | $id_M$,$id_M$ | S,S | $id_S$,$id_S$ | M,M | $id_M$,$id_M$ |

Verification of condition 5 consisted in checking that $G(g \circ f) = G(g) \circ G(f)$ for each pair of morphisms f and g in table 4. For example:

$$\begin{array}{ccc} f & \longrightarrow & G(f) \\ Hom_{Arch}(T,U) \times Hom_{SF}(U,R) & & Hom_{Arch}(M,M) \times Hom_{SF}(M,K) \\ <\varnothing,*> & & <id_M,*> \end{array}$$

$$\begin{array}{ccc} g & \longrightarrow & G(g) \\ Hom_{Arch}(R,T) \times Hom_{SF}(R,U) & & Hom_{Arch}(K,M) \times Hom_{SF}(K,M) \\ <id_R,*> & & <id_K,id_M> \end{array}$$

$$\begin{array}{ccc} g & \circ & f \\ Hom_{Arch}(R,T) \times Hom_{SF}(R,U) & & Hom_{Arch}(T,U) \times Hom_{SF}(U,R) \end{array}$$
$$Hom_{Arch}(R,U) \times Hom_{SF}(R,R)$$
$$<id_R,*>$$

$$\begin{array}{ccc} G(g) & \circ & G(f) \\ Hom_{Arch}(K,M) \times Hom_{SF}(K,M) & & Hom_{Arch}(M,M) \times Hom_{SF}(M,K) \end{array}$$
$$Hom_{Arch}(K,M) \times Hom_{SF}(K,K)$$
$$<id_K,*> . \qquad (3)$$

$$\begin{array}{ccc} g \circ f & \longrightarrow & G(g \circ f) \\ Hom_{Arch}(R,U) \times Hom_{SF}(R,R) & & Hom_{Arch}(K,M) \times Hom_{SF}(K,K) \\ <id_R,*> & & <id_K,*> . \end{array} \qquad (4)$$

Since (3) = (4), $G(g) \circ G(f) = G(g \circ f)$ is verified for this pair of morphisms. Similar calculations made on each pair of morphisms in Table 4 reveals that G verifies the condition 5.

## 5 Discussion

In discussion, I can propose the way of elaborating the bidirectional in order to preserve “architectural” information, i.e. the semantic relation between variables. I can propose to elaborate an additional class which is associated to the variables ones. => category is a support to provide the construction of the improvement of languages. Here I also have to explain the nature of equivalence between the two languages.

## 6 Related work

Here I have to describe the related works conducted in this area. This paragraphs should considers QQDODU – the object considered by transformations (generally semantic function: white box vs black box, etc) - the use of CT in MDA and transformations, etc – the level of abstraction (people generally considers M2, but one can wonders about the industrialization of such methodology). This part can be fulfilled with the last bibliography.

From introduction paragraph: Rule-based transformation systems [11] and triple graph grammars [12] are commonly adopted for migration. Both methods consist in identifying appropriate transformation rules and the rule matching algorithms [13]. In these methods, the transformation rules are incrementally identified and thus not automated. Moreover, preservation is evaluated *in fine*. Alternatively, other methodologies are quoted facing preservation. Concerning architecture, [14] proposed a method using process algebra. On the other hand, [15] adopted the category theory for preserving program maintainability in regard to the semantic function, i.e. the program computation [16]. Out of preservation, the category theory has already been applied to computer science to solve complex theoretical problems [17, 18].

## 7 Conclusion

UML enabled us to transcribe biological modelling languages using aggregation and association relationships between classes. This construction was possible on two contrasting languages commonly used in biology. Since our target was program migration, we only considered the terminal classes of the language and not inheritance. We obtained a mathematical representation of the language, where aggregation is expressed as inclusion and association as order relationship.

In our illustration, we established the functorial relationship APES $\longrightarrow$ SIMILE and the application is surjective. By construction, the functor provided both the object and morphism relationships. As each morphism corresponds to a translation rule, the construction of the migration function simply consists in instantiating the functor. According to [16], two programs are equivalent if they provide identical mapping between a set of inputs with a set of outputs. In the category theory, the concept of

equivalence is defined using the functor properties. Our construction supported that this definition of the equivalence of programs can be extended to architecture.
Conversely, SIMILE $\longrightarrow$ APES mapping is incomplete because of differences in architecture. With this example, we illustrated the advantages of applying the category theory to identify the aspects of migration which could not be preserved due to language expressiveness. In that case, conferring the properties necessary to verify the condition for functor establishment provides a support for language improvement. This aspect is not covered in this paper. Future work will consist in establishing the functorial bijection enabling bidirectional migration of the improved languages.

**Acknowledgments.** This publication was partially funded under the SEAMLESS integrated project, EU 6th Framework Programme for Research, Technological Development and Demonstration, Priority 1.1.6.3. Global Change and Ecosystems (European Commission, DG Research, contract no. 010036-2).

## 8 References

1. Sequeira, R.A., Olson, R.L., McKinion, J.: Implementing generic, object-oriented models in biology. Ecol. Model. 94, 17--31 (1997).
2. Von Bertalanffy, L.: Théorie générale des systèmes. Dunod, 2nd edition (2002).
3. Fritzson, P.: Principles of Object-Oriented Modelling and Simulation with Modelica 2.1. Wiley-IEEE Press (2004).
4. Costanza, R., Voinov, A.: Modelling ecological and economic systems with STELLA: Part III. Ecol. Model. 143, 1--7 (2001).
5. Muetzelfeldt, R, Massheder, J.: The SIMILE visual modelling environment. Europ. J. Agron. 18, 345--358 (2003).
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns – Elements of Reusable Object-Oriented Software (1st edition). Addison Westley Professional (2001).
7. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Software Eng. 26(1), 70--93 (2000).
8. Donatelli, M., Bellocchi, G., Carlini, L.: Sharing knowledge via software components: models on reference evapotranspiration. Europ. J. Agron. 24 (2), 186--192 (2006).
9. Mens, T., Van Gorp, P.: A Taxonomy of Model Transformation. Electr. Notes Theor. Comput. Sci. 152, 125--142 (2006).
10. Stevens, P.: Bidirectional model transformation in QVT: semantic issues and open questions. In: Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems, pp. 1--15. Springer, Heidelberg (2007).
11. Visser, E.: A survey of strategies in rule-based program transformation systems. J. Symb. Comput. 40(1), 831--873 (2005).
12. Königs, A, Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. Electr. Notes Theor. Comput. Sci. 148(1), 113--150 (2006).
13. Grunske, L., Geiger, L., Lawley, M.: A Graphical Specification of Model Transformations with Triple Graph Grammars. Lect. Notes Comput. Sci, 3748, 284--298 (2005).
14. Bernardo, M., Bonta, E.: Preserving architectural properties in multithreaded code generation. Lect. Notes Comput. Sci. 3454, 188--203 (2005).
15. Garcia AD., Haeusler, EH.: Code Migration and program maintainability - a categorical perspective. Information Processing Letters. 79 (5), 249--254 (2001).
16. Ward, M.P., Zedan, H.: Slicing as a program transformation. ACM Trans. Program. Lang. Syst. 29(2), n°7 (2007).

17. Barr, M, Wells, C.: Category Theory for Computing Science. Prentice Hall, Englewood Cliffs, NJ (1995).
18. Guo, J.: Using Category Theory to Model Software Component Dependencies. In: 9th IEEE International Conference on Engineering of Computer-Based Systems, pp. 185--194. IEEE Press, New York (2002).
19. Barr, M., Wells, C.: Toposes, Triples and theories. Reprints in Theory and Applications of Categories, 12, 1--288 (2005).
20. Mac Lane, S.: Categories for the Working Mathematician. Springer (1998)
21. Simulistics Ltd, http://www.simulistics.com
22. Forrester, J.W.: World Dynamics. Cambridge Mass. Wright-Allen Press (1971).
23. Agricultural Production and Externalities Simulator. http://www.apesimulator.it
24. Van Ittersum, M.K., Ewert, F., Heckelei, T., Wery, J., Alkan Olsson, J., Andersen, E., Bezlepkina, I., Brouwer, F., Donatelli, M., Flichman, G., Olsson, L., Rizzoli, A., Van der Wal, T., Wien, J.E., Wolf, J. Integrated assessment of agricultural systems - A component-based framework for the European Union (SEAMLESS). Agric. Syst., 98, 150--165 (2008).
25. System for Environmental and Agricultural Modelling; Linking European Science and Society, http://www.seamless-ip.org/
26. Szypersky, C., Gruntz, D., Murer, S.: Component software – beyond object-oriented programming. 2nd ed. Addison-Wesley, London (2002).
27. Hillyer, C., Bolte, J., van Evert, F., Lamaker, A.: The ModCom modular simulation system. Europ. J. Agron. 18, 333--343 (2003).
28. Hill, D.R.C.: Object-Oriented Analysis and Simulation. Addison-Wesley, Boston (1996).
29. Donatelli, M: Unpublished data. Task Leader of APES development, SEAMLESS EU 6 Framework Research Programme , (2007).