

# Exchanging Models Between Tools Supporting A Different Number of Modeling Layers

Hafedh Mili

Department of C. S., University of Québec at Montréal  
P. O. Box 8888, Downtown Station  
Montréal, Québec H3C 3P8, Canada

Francois Pachet

Sony Computer Science Laboratory  
6 Rue Amyot, 75005 Paris  
pachet@csl.sony.fr

## *Abstract*

**CDIF is a standard for representing models that allows CASE tools that do not support the same notation and/or the same semantics, to exchange models with a minimum loss of information, and a maximum level of accuracy. The CDIF does so by: 1) defining a language for defining notation (meta-meta model), and 2) prefixing model descriptions in a given notation by a definition of that notation; it is the responsibility of an importer to map the constructs of the exporter to its own. The mapping can be fairly straightforward for a wide range of variations between CASE tool notations, with minimum loss of information, but requires more work when the source and target notations support a different number of abstraction levels. We look at some of the issues involved, and propose along the way some transformation patterns.**

## **1. Introduction**

Generally speaking, we use the term *metamodeling* to refer to the practice of representing computational entities at more instantiation levels than the usual two, instance and class. By computational entity we mean pure data (entities), objects in the OO sense, tasks, processes, or any other computational artifact that can be created, modified or otherwise manipulated during the execution of a program. The need for metamodels is more frequent than one might first think, and has been practiced for some time. We identified three “typical” occurrences of

metamodeling, which we described as, 1) an abstraction of domain knowledge, 2) the representation of representation languages, and 3) an abstraction of computational behaviors [Mili & Pachet, 1998]. According to our categorization, the CDIF standard falls into the second category. CDIF is a standard for representing models that allows CASE tools that do not support the same notation and/or the same semantics, to exchange models with a minimum loss of information, and a maximum level of accuracy. The standard does so by: 1) defining a language for defining notations (meta-meta model), and 2) prefixing model descriptions in a given notation by a definition of that notation. In a model exchange transaction, it is the responsibility of the importer to map the constructs of the exporter to its own constructs.

The transformations from one modeling notation to another can be more or less straightforward, and more or less information lossless, depending on the differences between the underlying semantics, both in terms of coverage (e.g. one notation uses far more constructs than the other), and in terms of structuring (e.g. using a different semantic breakdown of information). The difference between the source and target notations may be reflected in the size (number of rules) and class of the required transformation grammar (e.g. a context-free versus a context-sensitive grammar), and the reversibility of the transformation, e.g. the extent to which  $T^{-1}oT(x)$  is different from  $x$ . In this paper, we are concerned with the mapping from a notation that uses domain metamodeling constructs to one that doesn't.

We first look at an instance of a domain model with several levels of abstraction. In section 3, we discuss transformation rules to a model that uses a single level. We conclude in section 4.

## 2. Metamodeling as abstraction of domain knowledge

Consider the example of a Savings & Loans Association whose member institutions offer different kinds of loans with various payment schedules. This is a simplified version of an actual modeling problem, for one such institution, on which the author has worked. Generally speaking, different types of loans have different payment formulas. The association between types of loans and payment formulas is practically industry-wide. For example, all student loans have an initial deferment period, after which regular payments must be made at a minimum pace, but a student may pay the remaining balance in full at any point in time. It is also almost always the case that mortgages are to be paid back according to some regular payments, and full payments usually carry a penalty<sup>1</sup> because lending institutions usually turn around and sell mortgages to potential investors with a guaranteed rate of annual return. For each one of these modes of payment, there are a number of parameters such as frequency of payment, ranges of allowable percentages, etc. Different member institutions of an S&L Association may offer different subsets of these ranges. For example, institution A will offer 6 month, 1, 2 and 3 year fixed-rate mortgages, while institution B will offer 1, 2, 3, and 5 year fixed-rate mortgages. An individual who walks into a particular institution (e.g. A) will get a mortgage with a *single value* from these parameters, e.g. a 3 year fixed-rate mortgage.

A first-cut object model for this problem looks like Figure 1. This model says that « a loan » is payable according to « a payment schedule », and that there are many types of loans, and many

types of payment schedules. What it fails to say is which schedules of payment are appropriate for which types of loans, *in general*, and *for individual institutions*. If we had one payment schedule per loan type, we could represent this easily within this model by specializing the association « Payable According » into three (or more) more specific associations. However, the combinations can be numerous, and when we take into account the specifics of individual institutions, they become unwieldy.

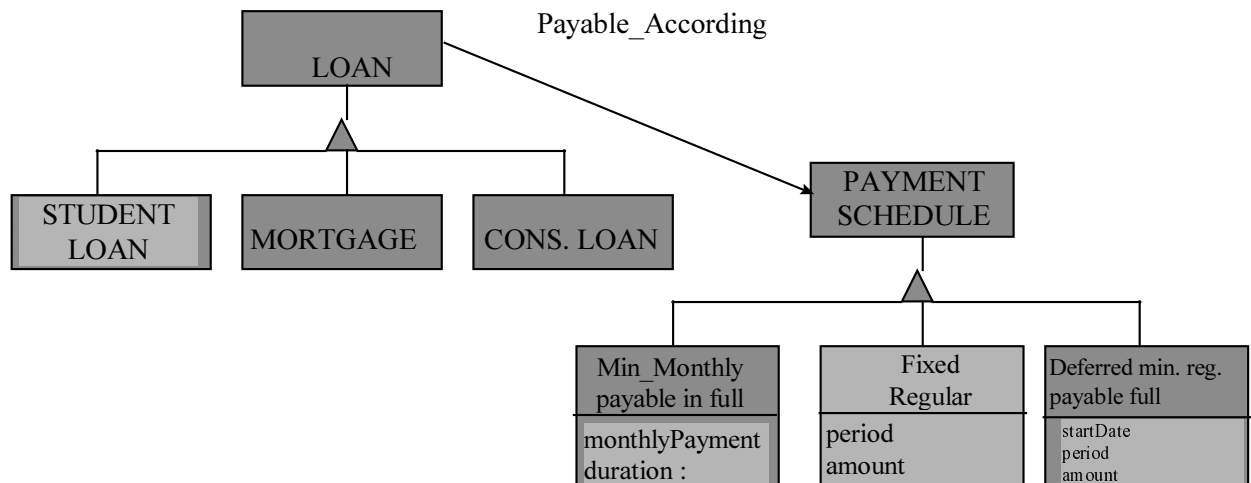


Figure 1. A simplistic model of loans and payment schedules.

What is really at issue here is the fact that we are attempting to represent information about two aspects, 1) about account *types*, and 2) about specific accounts of these types ; the latter have to abide by the constraints and parameters of the former. The object model of Figure 1 shows « a model of » the associations between *individual* accounts and *individual* payment schedules but very little about account types in general : if we restrict ourselves to a single institution, it may show which payment schedules apply to which accounts *for this particular institution*. We would need *a separate model for each institution*. The way of handling this consists of representing information explicitly about types of loans, types of payment schedules, and institutions, explicitly. Figure 2 shows such a model.

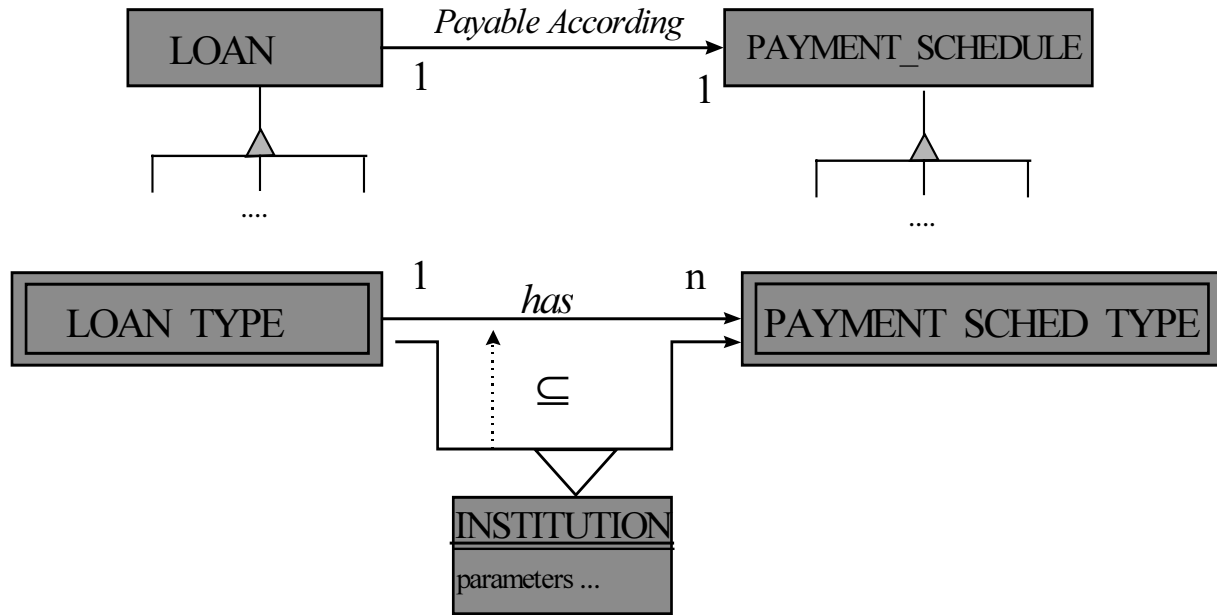


Figure 2. A complete model including a metamodel.

The upper part of the model is equivalent to the model of Figure 1. The lower part *is* the metamodel. The classes **LoanType** and **PaymentScheduleType** represent *types* or *classes* of loans and payment schedules, respectively, i.e. classes whose instances are themselves classes. The fact that a particular institution uses a specific subset of the set of payment schedules practiced by the industry at large is represented by the subset relationship ( $\subseteq$  in Figure 2) between the association that is specific to individual institution, and the general one. The data or knowledge that would have been embodied in separate models, one per institution, is now represented by a single (meta)model (lower part of Figure 2), and some specific instances of that model (e.g. a table). Figures 3a and 3-b show excerpts of the table that represent the meta data.

<sup>1</sup> Some financial institutions may allow payments of up to 10% of owed capital, annually, without penalty.

Account type	Payment schedule type
Mortgage	FixedReg., Deferred
SudentLoan	Deferred, PayableFull, FixedRegular
...	...

Figure 3-a. Representing industry-wide data.

Account type	Branch	Payment schedule type
Mortgage	B123	FixedRegular
StudentLoan	B122	Deferred, PayableFull
...	...	...

Figure 3-b. Representing branch data.

The *explicit* presence of this information is *also* valuable for the purposes of coding the desired behavior concisely and generically.

### 3. Exchanging models and metamodels with CDIF

#### 3.1 The exporter's point of view

To represent the models in Figure 2, along with the corresponding tables, we need to define the following constructs:

- An *EntityType* which will represent **AccountType** and **PaymentScheduleType**
- An *Entity* which will represent **Account**, **Mortgage**, etc., along with the various payment schedule types, such as **FixedRegular**, **Deferred**, etc.
- Either define a *GeneralAssociation* which can link instances of *Entity* among each other, and instances of *EntityType* among each other, or even instances of *Entity* to instances of *EntityType*, or define two (or three) types of associations, and
- Some sort of an *InstanceType* of the corresponding associations to communicate the data inherent in the two tables.

Once these structures have been defined, then we can send the model as a stream of constructs looking like<sup>1</sup>:

```
// definition of metamodel entities, i.e. EntityType, Entity, Attribute, Association, etc.
METAOBJECT MetaEntity Name = EntityType ...
METAOBJECT MetaRelation Name= GeneralAssociation ...
...
// definition of model
METAENTITY EntityType Name=AccountType Id = ET102 ...
METAENTITY Attribute Name= AccountTypeName onEntity=AccountType ...
METAENTITY Entity Name=Account Id=E112 ...
METARELATION GeneralAssociation Name= Specialization Id= GA23
METAENTITY Role Name=Superclass relation=Specialization
                                metaEntity=Entity | EntityType
METARELATION GeneralAssociation Name= Membership Id= GA34
GeneralAssociation Membership member= Account class=AccountType
...
// definition of data
RelationInstance PaymentScheduleForIndustry accountType=StudentLoan
                                PaymentScheduleType=FixedRegular

RelationInstance PaymentScheduleForIndustry accountType=StudentLoan
                                PaymentScheduleType=Deferred
...
```

In addition to this structural information, the exchange format might include graphical information about the layout of the model and the domain metamodel shown in Figure 2.

There is some confusion in this model as to whether we should consider the specialization relationship as an atomic/terminal construct of the source language, at the same level as *Entity* or *EntityType*, or whether it is to be considered as a specific instance of a general association

---

<sup>1</sup> The author is not familiar with the CDIF meta-meta-model, and the names or constructs shown do not reflect the contents of the standard.

between meta entities, which includes the representation of arbitrary associations between entities (e.g. *WorksFor*) or the representation of arbitrary associations between entity types (*Specialization*, *Membership*, *Partition*, etc.). In the above example, we assume the second interpretation. If we look at this model on the receiving end, *Membership*, *Specialization*, and any kind of association (e.g. a person *WorksFor* a company) might well look alike, because the actual distinctions are not syntactic, but rather semantic. For instance, what it means for a class/type to be a subclass/subtype of another is *behavioral* in the sense that it dictates how parts of the model are affected by changes in the related parts (e.g. inheritance of attributes, for the case of specialization). The same goes for membership (e.g. the “applicability/reification of attributes”).

CDIF does make a provision for the subtype relation, which is part of the meta-meta-model, and its semantics are understood across, but we are not certain that the semantics of the membership relation, which is implicit to the meta-meta-model, are recognized *explicitly* as relationships that need to be expressed and manipulated.

There is some question as to whether the actual relationship should be called “membership” or “is modelled by”. The first is purely *extensional* (membership) while the second relates an individual to an *intensional* representation of its class [Mili & Pachet, 1998].

### **3.2 The importer’s point of view: metamodeling patterns**

Whichever way we choose to convey the metamodeling dimension in the exchanged model to the importer, the latter has to find a faithful and operational model in the target language.

In the remainder of this section, we look at a couple of analysis/design patterns that enable us to represent metamodels in languages/environments that support only two levels of abstraction.

#### **3.2.1 A basic pattern : representing two levels of instantiation**

Figure 4 shows the object model of a design pattern we used many times to represent class information explicitly during run-time. Implementation-wise, instances of **ObjectType** have two dictionaries, one to hold descriptions of attributes (accessible by attribute name, or ‘attName’), i.e. instances of class **Property**, and one to hold components, i.e. other instances of the class **ObjectType**, accessible by component role name (e.g. ‘leftLeg’ and ‘rightLeg’, both of which could be instances of **Leg**). An actual object will be represented by an instance of the class **Object** or one of its subclasses. Each object (instance of **Object**) will also have two dictionaries, one to hold attribute values (instances of **PropertyValue**, accessible by ‘attName’), and one to hold components (instances of **Object** or one of its subclasses, accessible by ‘roleName’). Objects point to the instance of **ObjectType** that describes them. The class **Object** (and its subclasses) will implement a constructor that takes an instance of **ObjectType** as an argument, from which to initialize the dictionaries. If the classes of objects in the application have no distinguishing behavior except for the component and attribute access methods, all the classes can be represented by a single class, **Object**.

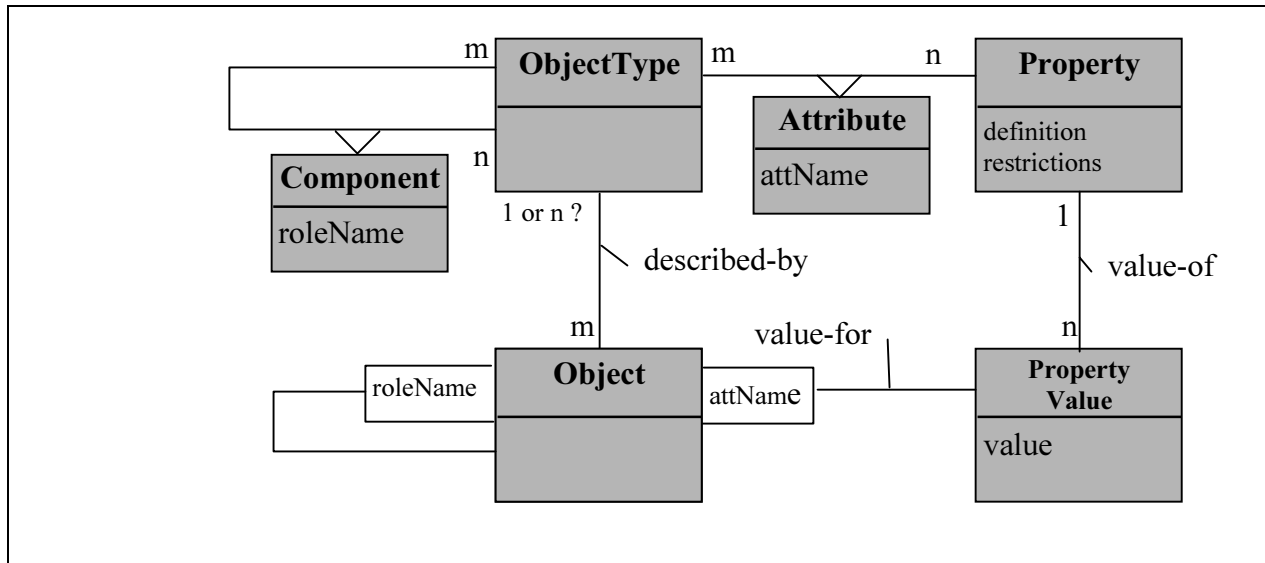


Figure 4. Simulating meta-types in an environment that does not support them.

This pattern has proven useful for cases where we have to deal with several classes with complex data structures but whose behavior is no, or little, more than structure access : all the access methods can be coded generically as table access methods. It supports the addition of new attributes to existing classes during time, and the addition of new classes altogether. One application area in which this pattern was used was a CAD application that manipulated 3-D objects drawn by the user with a drawing editor. The second application had to do with the processing (analysis and routing) of messages in an avionics message processing system for a major US Airline, where incoming messages have unpredictable structure and contents.

### 3.2.2 The Zig-Zag pattern : implementing multi-level metaclasses.

We owe this pattern to Sahraoui and Revault ([Sahraoui,1995], [Revault&Sahraoui,1995]) and the MétaGen system. MétaGen is a Smalltalk-based CASE tool generator that generates tools that transform models in a source description language (e.g. analysis model) to models in a target description language (e.g. design or implementation model). MétaGen takes as input a description of a source language, a description of a target language, and set of rules for transforming source language constructs to target language constructs. It outputs a graphical editor for the source language, a transformation procedure, and a graphical editor for the target language [Sahraoui,1995].

Although Smalltalk supports metaclasses, it supports only one level of metaclasses, and metaclasses have single instances. MetaGen requires several levels of instantiation which cannot be accommodated by Smalltalk's instance → class → metaclass chain. The ZIG-ZAG pattern is illustrated in Figure 5. This pattern may be used with tools that support two levels of instantiation.

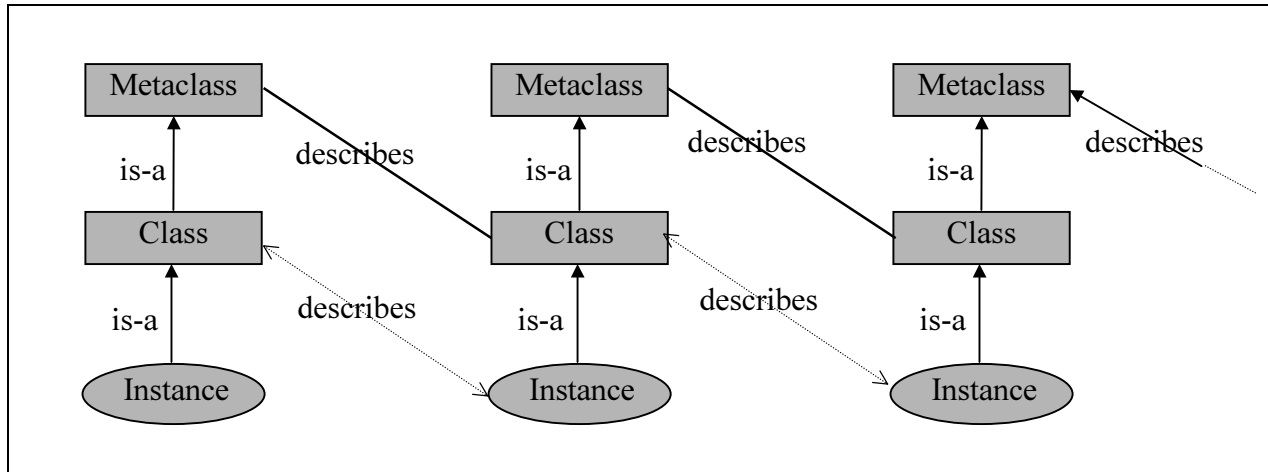


Figure 5. The ZIG-ZAG pattern.

## 4. Discussion

We identified three “typical” occurrences of metamodeling, which we described as, 1) an abstraction of domain knowledge, 2) the representation of representation languages, and 3) an abstraction of computational behaviors [Mili & Pachet, 1998]. According to our categorization, the CDIF standard falls into the second category. CDIF is a standard for representing models that allows CASE tools that do not support the same notation and/or the same semantics, to exchange models with a minimum loss of information, and a maximum level of accuracy. In a model exchange transaction, it is the responsibility of the importer to map the constructs of the exporter to its own constructs. In this paper, we took a closer look at the problem of mapping models that support different levels of abstraction for domain knowledge, to ones that support the usual two (instance and class). Because CDIF supports only limited semantic specification of metamodeling constructs (i.e. modeling languages), we suspect that special constructs may need to be added to the meta-meta-model with agreed upon semantics. We also suspect that while that ensures the accuracy of the exported model, importers that do not support domain metamodeling need to apply analysis/design patterns for supporting several levels of abstraction. In [Mili&Pachet,1998], we discussed some of the behavioral aspects related to the application of the design patterns (e.g. how class methods may be handled in a classless languages). A lot of work remains to be done to translate these guidelines into reliable transformation rules that may be applied to some of the operational/behavioral models covered by CDIF (e.g. the data flow modeling and state/event subject areas).

### *Acknowledgements:*

Over half the material in this position paper is taken from a (better written) paper authored with François Pachet (see reference [Mili & Pachet, 1998] below), which, itself, benefited from discussions at a 1995 OOPSLA workshop on metamodeling co-organized by the author (see [Mili et al., 1995]), and subsequent discussions with Jean Bézivin and Jim Odell, in person, or moderated through the OA&DTF mailing list.

## 5. References

- [Bezivin,1995] Jean Bézivin, ``Technologie objet et ingénierie des besoins : une réconciliation nécessaire,’’ *L'Objet*, vol. 1, n 1, p. 21-26, 1995.
- [Blaaha & Premerlani, 1994]
- [Briot & Cointe,1989] Jean-Pierre Briot and Pierre Cointe, ``Programming with Explicit Metaclasses in Smalltalk-80,’’ *OOPSLA '89*, New Orleans, 419-432.
- [Mili et al., 1995b] Hafedh Mili, François Pachet, Ilham Benhyaya, and Frederick Eddy, ``Report on the OOPSLA'95 Workshop on Metamodeling,’’ *Addendum to the OOPSLA'95 proceedings*, ACM SIGPLAN notices.
- [Mili & Pachet, 1998] Hafedh Mili and François Pachet, Ilham Benhyaya, ``Metamodeling design patterns’’, submitted to special issue of *Information Systems: An International Journal* on metamodeling, April 1998.
- [Missaoui et al.,1998] Rokia Missaoui, Houari Sahraoui, and Robert Godin, `` Migrating to an Object-Oriented Database Using Semantic Clustering and Transformation Rules ,’’ to appear in *Knowledge and Data Engineering*, 1998.
- [Revault & Sahraoui, 1995] Nicolas Revault and Houari Sahraoui, ``A Metamodeling Technique : The MétaGen System,’’ in *Proceedings of TOOLS Europe '95*, Versailles, France, 1995.
- [Sahraoui,1995] Houari Sahraoui, *Application de la méta-modélisation à la génération des outils de conception et de mise en oeuvre de bases de données*, Doctoral Thesis, University of Pierre & Marie Curie (Paris 6), 1995.