Proposals for Multiple to Single Inheritance Transformation

Michel Dao[a], Marianne Huchard[b], Thérèse Libourel[b], Anne Pons[c], Jean Villerd[b]

[a]France Télécom R&D DTL/TAL
38-40 rue du général Leclerc, 92794 Issy Moulineaux Cedex 9, France
[b]LIRMM – CNRS et Université Montpellier II – UM 5506
161 rue Ada, 34392 Montpellier Cedex 5, France
[c]Dept Informatique – Université du Québec à Montréal
C.P. 8888 – Succursale centre ville, Montréal Québec H3C 3P8, Canada

ABSTRACT

We present here some thoughts and ongoing work regarding transformations of multiple inheritance hierarchies into single inheritance hierarchies. We follow an approach that tries to categorize multiple inheritance situations according to a semantic point of view. Different situations should be captured through diagrammatic UML annotations that would allow to detect a given situation and hence apply the appropriate transformation, automatically if possible.

Keywords: multiple inheritance, UML, model transformation

1. INTRODUCTION

Multiple inheritance (MI) vs. single inheritance (SI) was the subject of numerous passionate discussions during the era of the emergence of object-oriented programming languages. Those discussions have more or less come to an end and we are left with: not so badly implemented MI languages (e.g. Eiffel and CLOS) that are scarcely used, widespread not so well implemented MI languages (e.g. C++) or the flagship language Java (but for how long?) with SI (and MI for interfaces).

Lately, a shift in software development process has given an increased importance to modelling, specially through the widespread use of UML.[1] Of course UML proposes MI: as opposed to programming languages, there are no conflicts to be resolved at compile time when using MI in UML and those who believe, as we do, that MI can be a good means of modelling existing entities can use it without caution. Furthermore, UML proposes annotations allowing inheritance links to convey special meanings.

More recently, Model Driven Architecture (MDA)[2] has proposed a framework in order to formalize the extensive use of models during the software development process. MDA fosters the use of different models throughout the process, the models of one phase of development being derived from the models of a previous phase. More specifically, a Platform Independent Model (PIM) may be used to generate a Platform Specific Model (PSM). For instance, a UML design level static model may be used to generate source code in a given programming language. Our proposition fits into this precise scheme: how can we automate the transformation of a MI UML class diagram into a SI class diagram, hence allowing straightforward transformation into SI programming language?

There exist several works on the subject of MI vs. SI.[3–5] In a previous project in which we participated, two approaches have been considered that may eventually be combined. The first one may be described as "combinatorial" and consists in defining a strategy to remove inheritance links so as to minimize the number of properties (attributes and methods) that it is necessary to duplicate. Such a strategy can be based on a set of metrics that allow to measure *a priori* the impact of the deletion of an inheritance link. A first work following this approach has been realized[6] that yielded interesting results but needs to be refined and completed in order to be fully usable.

Another approach, which is the subject of this article, may be described as "semantic". The idea is to consider that MI may appear in several typical situations that correspond to different semantics and that for each situation there may exist several possible transformations into SI. The problem of MI to SI transformation may therefore be decomposed as follows:

- elaborate a list of the different situations of MI and of the corresponding possible transformations into SI;
- be able to find occurences of those different situations in a class hierarchy;
- be able to apply the pertinent transformation.

The structure of inheritance is clearly not sufficient to determine a situation of MI. UML standard proposes some annotations of inheritance and we believe that those annotations may help in spotting specific inheritance situations but they are limited and do not allow to capture all situations. In a previous article,[7] we have proposed some extensions to those annotations in order to enhance the semantic expressiveness of inheritance links in UML class diagrams. Furthermore, some other informations (size of classes, size of generalization set, need of symmetry, number of inherited methods, etc.) may help to determine the best transformation to be applied.

We first present the annotations that may be used to convey semantic inheritance information in UML class diagrams. This is followed by a proposition of a set of MI to SI transformations that we have gathered in existing work (and have partly adapted). Then we propose a tentative list of typical MI situations associated with one or more pertinent transformations. We conclude by discussing our approach and its perspectives.

2. MULTIPLE INHERITANCE ANNOTATIONS

A first type of UML annotation[1,8] is the *discriminator* that allows one to group subclasses into clusters that correspond to a semantic category. For instance, in Figure 1*, class `Employee` is specialized according to two criteria, `:status` (related to the salary payment), and `:pension` (vested vs unvested). Discriminators involve a partition of the specialization links coming to a parent class: in our example this partition has two elements, the set of the links labelled with the discriminator `:status` from one side, the set of the links labelled with the discriminator `:pension` from the other side.

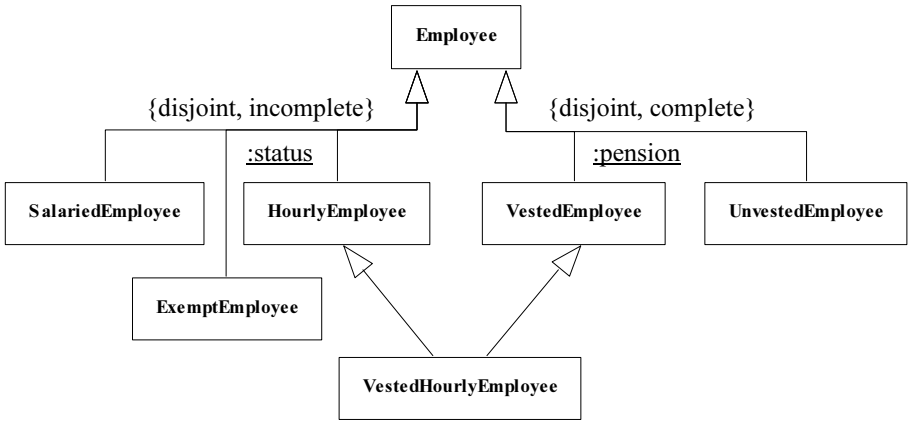

Figure 1. Exemple of annotated multiple inheritance

A second kind of annotation existing in the UML notation appears as constraints about the extension (instance set) of a class C and of its subclasses. We denote by E the set of the direct subclasses which are gathered by such an annotation. Four constraints are predefined:

overlapping an instance of C can simultaneously be instance of several classes of E;

disjoint an instance of C is instance of at most one class of E;

complete elements of E are origins of links annotated by a same discriminator; any instance of C is instance of one of the elements of E;

*This example is borrowed from.[9]

incomplete the classes of E are origins of links annotated by a same discriminator; an instance of C is not necessarily instance of one of the classes of E.

We have proposed to extend this set of annotations with the following[7]:

alternative the characteristics of the super classes are used alternatively as in the case of an amphibian vehicle;

concurrent (special case of overlapping for roles and states): father/husband;

successive (special case of disjoint with a temporal scheduling): chrysalis/caterpillar/butterfly, child/teenager/adult;

exclusive (special case of disjoint for roles and states): empty/full for a stack, married/single for a person;

repeated similar to repeated inheritance in C++: a property may be inherited along several different paths from the same indirect super class;

combined when a class is directly specialized according to several criteria denoted by discriminators, an instance may be constrained to belong to at least one class of each discriminator;

disjoint partition conversely, an instance may be constrained to belong to only one discriminator;

implementation in numerous examples of multiple inheritance in the programming area, a class ends up deriving from superclasses that it specializes for implementation needs.

3. INHERITANCE TRANSFORMATIONS

Figure 2 shows an initial MI situation and five possible transformations into SI.

Transformation 1 – Duplication The first transformation that may be applied is to remove one of the inheritance links and to duplicate in the subclass all the properties that were inherited through this link. The advantage of this solution is simplicity but duplication of code is always a bad thing regarding reuse and maintenance. The choice of the inheritance link to cut could be based on the number of properties that must be duplicated: the less, the better.

Transformation 2 – Nested generalizations This transformation consists in cloning one set of classes corresponding to a discriminator (here `:discrCD`) into subclasses of each class corresponding to the other discriminator. This transformation may only be useful when there are few classes under the chosen discriminator and it may be difficult to choose the discriminator to clone. Furthermore, the naming conflicts produced by the new classes must be resolved but polymorphism is kept.

Transformation 3 – Direct link This transformation can be seen as a double duplication: both inheritance links are cut and the properties from both superclasses (`B` and `C`) are duplicated into class `E`. This transformation involves more duplication than the duplication involved in transformation 1 but allows to preserve the symmetry of the class hierarchy if this is relevant.

Transformation 4 – Role aggregation Another solution is to transform one of the inheritance links into an aggregation[†] link. Polymorphism is replaced by a delegation mechanism at the expense of the creation of a new class `A/CD` and of code rewriting. In our example each method or accessor of class `C` should be replaced by one with the same name in class `A` performing a call to the right method or accessor in class `C`. The choice of the inheritance link to be replaced by delegation could be based on the amount of properties to be redefined or one could choose the class belonging to a discriminator that is complete because such a replacement would be done once and for all.

[†]In fact, that might be a composition link.

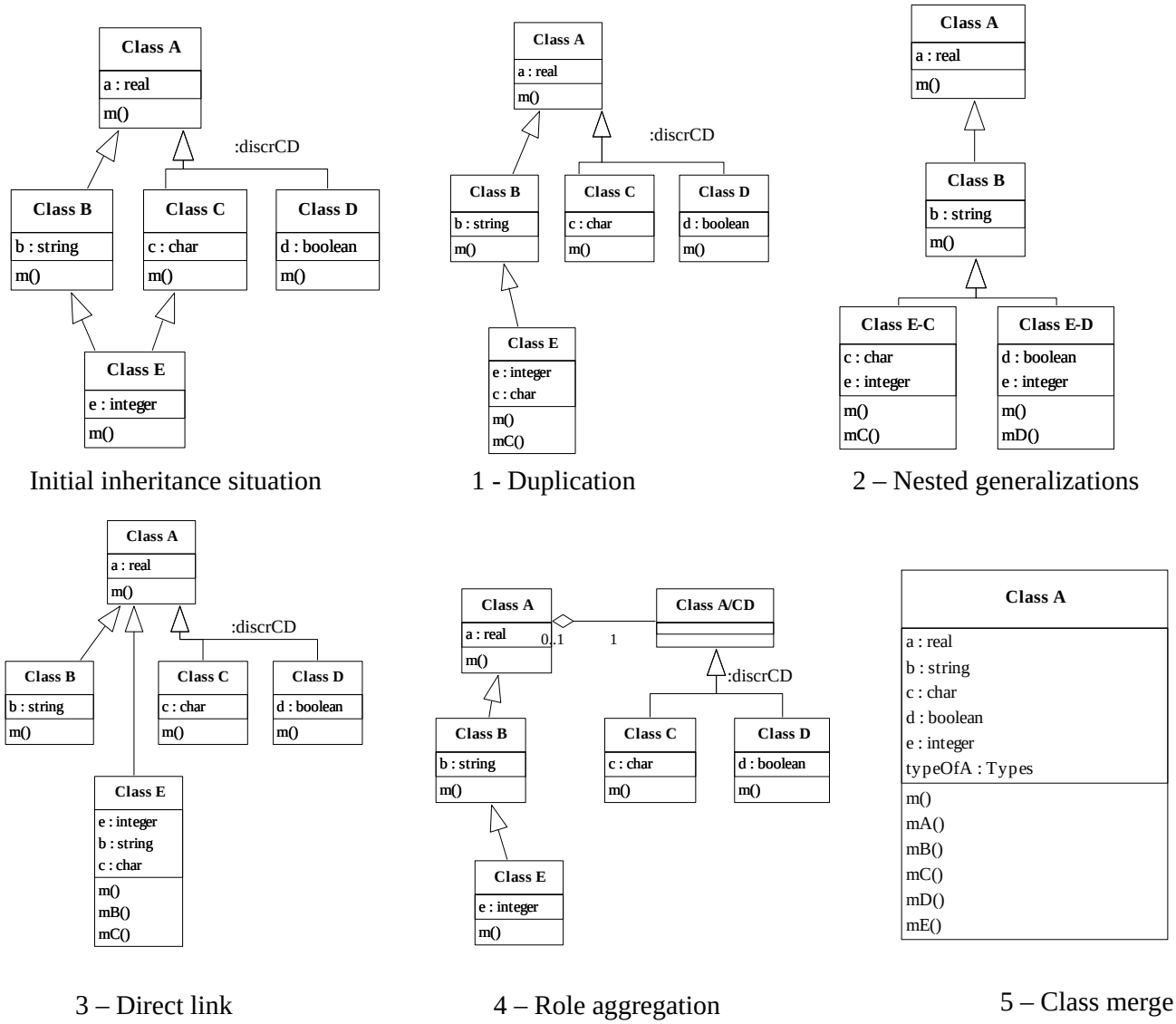

Figure 2. Propositions of inheritance transformations

Transformation 5 – Class merge This transformation merges `A` and all its subclasses into one unique class. Conflicting properties must be renamed and a dispatch mechanism must be implemented that uses a typing attribute indicating the type of instances of this unique class.

Transformation – Interfaces Another transformation not depicted in the figure consists in defining a MI interface hierarchy corresponding to the MI class hierarchy and to establish implementation links between the class hierarchy and the implementation hierarchy.

Our ongoing work is to define a mapping between a given MI situation and one (or several) pertinent transformations. Table 1 lists a first attempt of such a mapping. This is clearly an incomplete and debatable table that needs completion, refinement and discussion. Let us discuss a couple of our proposals.

First, the *complete* annotation implies that no other classes should be added to the cluster of classes gathered under this annotation, typically a discriminator. Therefore, in the case where there are only few classes involved, one multiple inheritance cluster could be transformed using the *nested generalization* transformation. In this

case, this would not lead to a too complex diagram with a great number of classes that are not all useful. Another case is when there is only one inheritance link that originates from a *complete* annotation, one could use the *role aggregation* transformation with this inheritance link being replaced by the aggregation link.

The *combined* annotation stipulates that a subclass inherits from at least one class from each discriminator. In the case of the *alternative* sub case, this argues in favor of the *direct link* transformation where all superclasses are treated equally.

The *implementation* annotation can be viewed as a conceptual model of the interfaces of Java (*able: cloneable, serializable, etc.) and therefore the most natural transformation consists in using interfaces to represent multiple inheritance in that case.

Situation	Semantic subsituation	Transformation	Comment
overlapping	concurrent	role aggregation	
disjoint	successive exclusive	role aggregation	
complete		nested generalization role aggregation	few classes under the chosen discriminator and at most two discriminators complete = will not evolve
combined	alternative	role aggregation nested generalizations direct link	few classes under the chosen discriminator
repeated		role aggregation	
implementation		interfaces	

Table 1. MI situations and transformations

4. DISCUSSION AND PERSPECTIVES

We have presented here our ongoing work on MI to SI transformation based on semantic annotations of UML class diagrams. We have so far enriched UML annotations with some new ones and determined a set of transformations. We are currently studying the mapping between a given situation of inheritance (UML extended annotations and other criteria) and the possible transformations that may be applied.

It is obvious that such transformations should be applied to a class hierarchy as automatically as possible. We have realized a limited implementation of two of the transformations listed in Section 3 in UML CASE tool Objecteering[‡] using its proprietary object-oriented language J. We are wondering if this type of procedural implementation is best suited for our purposes. As inheritance transformation may be seen as model transformation, we are considering the use of a model transformation language (such as those for which OMG is requiring for proposals) to express both the research of MI inheritance situations and their transformations into SI.

We believe that the work we have presented here may be an incentive for the following discussion topics:

- can we reconcile multiple and single inheritance by allowing the latter to be (partially) automatically obtained from the former?
- does this type of transformations fit into the MDA approach?
- to which extent can we classify multiple inheritance into well defined semantic categories?
- to which extent can we capture those semantic categories in UML annotations?

[‡] `www.objecteering.com`

REFERENCES

1. U2 Partners, *Unified Modeling Language: Superstructure, version 2.0, 3rd Revised submission to OMG RFP ad/00-09-02, http://www.omg.org/cgi-bin/doc?ad/20 03-04-01*, april 2003.
2. Object Management Group, *MDA-Guide, V1.0.1, omg/03-06-01*, june 2003.
3. K. Thirunarayan, G. Kniesel, and H. Hampapuram, "Simulating Multiple Inheritance and Generics in Java," *Computer Languages* **25**(4), pp. 189–210, 1999.
4. M. Malak, "Simulating Multiple Inheritance," *Journal of Object-Oriented Programming* , pp. 3–5, april 2001.
5. Y. Crespo, J.-M. Marquès, and J. Rodriguez, "On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies," in *Proceedings of the Inheritance Workshop at ECOOP 2002*, Black, Ernst, Grogono, and Sakkinen, eds., pp. 30–37, 2002.
6. C. Roume, "Going from Multiple to Single Inheritance with Metrics," in *Proceedings of the sixth ECOOP workshop on Quantitative Approaches in Object Oriented Software Engineering (QAOOSE 2002)*, F. Brito e Abreu, M. Piattini, G. Poels, and H. Sahraoui, eds., pp. 30–37, 2002.
7. M. Dao, M. Huchard, T. Libourel, and A. Pons, "Extending the Notation for Specialization/Generalization," in *Proceedings of MASPEGHI'03, ISBN 2-89522-035-2*, pp. 61–67, (CRIM, Université de Montréal), 2003.
8. Rational Software Corporation, *UML v 1.3, Notation Guide*, version 1.3 ed., june 1999.
9. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object Oriented Modeling and Design*, Prentice Hall Inc. Englewood Cliffs, 1991. pages 15–84.