Metamodel Reuse with MOF

Xavier Blanc, Franklin Ramalho[1] and Jacques Robin[2]

Laboratoire d'Informatique de Paris 6, Université Pierre et Marie Curie (LIP6-UMPC)
8, Rue du Capitaine Scott, 75015 Paris, France
`xavier.blanc@lip6.fr, franklin.ramalho@gmail.com, robin.jacques@gmail.com`

Abstract. As model-driven development promotes metamodels as key assets it raises the issue of their reuse throughout a model-driven product line life cycle. One recurrent reuse need occurs when metamodeling integrated multi-language platforms: one construct from one language is integrated to constructs from other languages by *generalizing* it, making it more expressive. None of the metamodel assembly facilities provided by MOF and UML (`import, merge` and `combine)` or others proposed in previous work adequately addresses this need. We thus propose a new `reuse and generalize` facility for such purpose.

1 Introduction

Model Driven Development (MDD) raises the level of abstraction of the development life cycle by shifting its emphasis from code to models, metamodels and model transformations. It views any software artifact produced at any step of the development process as a valuable asset by itself to reuse across different applications and implementation platforms so as to cut the cost of future development efforts. Since they drive much of the MDD process, metamodels are the first artifact to reuse when an MDD team extends its portfolio of application domains, product requirements or target implementation platforms. However, the issue of metamodel *reuse* has not yet received much attention in the MDD literature. The reuse facilities provided by standards such as MOF 2.0 [11] and UML 2.0 [12] [13] or proposed in previous research on the topic [5][6][9][10] are essentially limited to two simple reuse needs. The first is *reuse as is* through inter-package visibility facilities such as package `import` in MOF or copy and paste transformations such as package `combine` in MOF. The second is *reuse as specialization* through inheritance facilities such as package `merge` in UML 2.0. As larger, more complex metamodels get constructed and then incrementally extended during their lifecycle, a diversity of

[1] Now at Departamento de Sistemas de Computação, Universidade Federal de Campina Grande (DSC-UFCG) Campina Grande, PB, Brazil and Centro de Informática, Universidade Federal de Pernambuco (CIn-UFPE), Recife, PE, Brazil.

[2] Now at Centro de Informática, Universidade Federal de Pernambuco (CIn-UFPE), Recife, PE, Brazil.

L. Briand and C. Williams (Eds.): MoDELS 2005, LNCS 3713, pp. 661-675, 2005.
© Springer-Verlag Berlin Heidelberg 2005

more sophisticated metamodel reuse needs is emerging. They prompt the definition of new metamodeling assembly facilties to address them and proposals to incorporate these facilities in MOF.

In this paper, we identify one such more sophisticated metamodel reuse need that we call *reuse and generalize*, and we propose a new metamodeling assembly facility that addresses it. This need is pervasive for various classes of metamodeling tasks. One such class is metamodeling multi-paradigm languages that historically result from successive extensions of a mono-paradigm core with constructs imported from other paradigms and adapted from integration. Another such class is metamodeling integrated multi-language platforms that support development in several languages that are conceptually integrated (as opposed to just loosely coupled through communication interfaces). In such cases, constructs from different languages have been extended or integrated by *generalizing* them so as to subsume constructs from other languages. Historically, such situation occurred many times over for example when defining concurrent or object-oriented extensions of imperative, functional or logic programming languages. It is currently occurring again as distributed, XML-based extensions of many languages are being defined.

The paper is organized as follows. In the next section, we advocate a *compositional* approach to metamodeling multi-paradigm languages or integrated multi-language platforms. With such an approach, metamodels of minimal size with potential for reuse are first built separately and then assembled in a way that mirrors how constructs from each paradigm or language are reused in the integrated whole. We identify and discuss five general benefits of such an approach in subsection 2.3. To make our points concrete, we illustrate these benefits, as well as all the subsequent points we make throughout the paper, on a specific case study: metamodeling the Flora platform [18]. Prior to discussing these points, we thus have to introduce the necessary background on Flora and then motivate having chosen it for the case study. We respectively do so in subsections 2.1 and 2.2. Briefly, metamodeling Flora is an interesting illustrative case study because it supports in an integrated way several languages and paradigms and it is a versatile platform that supports programming, meta-programming, reasoning, meta-reasoning as well as data and metadata definition, updates and queries.

Since Flora is such a comprehensive platform that results from a long, multi-step integration process, starting from section 3, we focus the scope of our illustrative examples on only *one* such step: the integration of first-order logic programming in Prolog with meta-programming based on high-order variables, resulting in HiLog. Subsections 3.1 and 3.2 thus respectively present simplified metamodels of these two languages supported by Flora. Subsection 3.3 then further zooms on only the relevant elements of theses two metamodels that we use to illustrate the general need to *reuse and generalize* metamodel elements. This example involves reusing elements metamodeling first-order predicates to metamodel their generalization as high-order predicates. In the subsequent subsections 3.4 to 3.6 we explain in detail why none of the three package assembly facilities provided the MOF 2.0 and UML 2.0 standard (`import, merge` and `combined`) can be used to address such reuse need. In subsection 3.7, we propose a fourth package assembly specifically designed for such need that we thus call `reuse and generalize`. In section 4, we review the

literature in metamodel reuse and show that none of the proposals put forward to date address such a need. In section 5, we conclude by summarizing our contributions.

2 Metamodeling Integrated Multi-language Platforms

In this section, we first present the main characteristics and historical genesis of the Flora multi-language, multi-paradigm and multi-purpose platform. Metamodeling Flora is the larger case study from which we extracted the examples that we use throughout the paper to illustrate the general points that we make on metamodel reuse. We then motivate the choice of such metamodeling task as illustrative case study. We take the opportunity to point out the mutual synergy that exists between MDD and platforms integrating multiple concepts and services such as Flora. Finally, we advocate a compositional approach to metamodeling such platforms by identifying five benefits that it provides and illustrating each of them on the Flora metamodeling case study.

2.1 Flora: An Integrated Multi-language, Multi-purpose Platform

Flora implements a subset of the language Concurrent Transaction Frame Logic (CTFL). Syntactically, CTFL integrates constructors from: (a) logic programming, (b) object-oriented programming, (c) imperative programming and (d) concurrent programming. Semantically, it provides declarative formal accounts of all these constructs by way of logical model theories. Historically, CTFL and Flora result from a 15 year research effort to overcome the failings of ISO Prolog to fulfill the original logic programming ideal: a language with declarative logical semantics that is simultaneously (a) a practical Turing-complete *programming* language, (b) an expressive *knowledge representation* language and (c) a concise *data definition, update and query* language. From its root in Prolog, CTFL evolved incrementally, as a series of largely *orthogonal* extensions, each one providing a semantically well-founded logical alternative to the extra-logical predicates of Prolog that betrayed the original logic programming ideal. This evolution had six main steps:

1. Extending Prolog with the logically *Well-Founded Negation as Failure* (WFNAF) connective;
2. *HiLog (HL),* extending Prolog with high-order syntax inspired from functional programming for meta-level programming, reasoning and querying but with first-order logical semantics [4]
3. *Transaction Logic (TL),* extending Prolog with logically well-founded backtrackable knowledge base updates and procedural constructs such as conditionals and loops [2];
4. *Frame Logic (FL),* extending HiLog with an object-oriented syntax [8] and logical semantics for single-source multiple structural and behavioral inheritance integrated with deduction [17];
5. *Concurrent* Transaction Logic *(CTL),* extending Transaction Logic with multiple threads, critical sections and inter-thread messages [3].

The Flora platform implements extensions 1-4 above, *i.e., Sequential Transaction Frame Logic* (STFL). It does so reusing the XSB Prolog platform that already implements extensions 1-2. Flora has two main components: the Flora Compiler that transforms an STFL program onto an optimized XSB Prolog program, and the Flora Shell, a front-end for queries in STFL that transforms STFL queries into semantically equivalent XSB queries, calls XSB to answer them and passes the answers back to the user in STFL syntax.

2.2 The Synergy Between MDD and Integrated Multi-language Platform

From an MDD perspective, a platform such as Flora that is multi-language, multi-paradigm and multi-purpose is very interesting in that it raises the possibility to rely on a single platform to (a) *run* a prototype to *validate* functional requirements of an application with the users, (b) *run* the same prototype to *test* its correctness on a set of specific cases, but also (c) *query* the same prototype to formally *verify* general properties that abstract from any set of specific cases, thus providing stronger robustness guarantees. Performing such verification on an implementation allows circumventing the major loophole of traditional formal development that relies on two different languages (one formal but non-executable for modeling and one executable but with no clear semantics for programming) and two different platforms (one for model validation and verification, and one for implementation execution and testing): namely the reintroduction of semantic errors while programming a verified model. We are currently exploring this possibility in the on-going MODELOG project [15] which investigates the development of a CASE tool for MDD providing a variety of services. Chief among these services is the fully automated generation of Flora code from UML models that consist of OCL annotated class and activity diagrams linked together through object flows. In order to develop this CASE tool for MDD using MDD itself, the first two steps of the MODELOG project are (1) developing a metamodel of the target platform Flora and (2) specify model transformations as QVT[14] relations between elements of this target metamodel and the source UML and OCL metamodels made available by the OMG. It is while carrying out the first task that we identified and addressed the metamodeling reuse issues presented in this paper.

2.3 Compositional Metamodeling and Its Benefits

For metamodeling languages and platforms that tightly integrate multiple paradigms and purposes, we advocate a *compositional approach* that faithfully mirrors their structure as a simple core and a set of largely orthogonal and complementary extensions. With such an approach, one starts by independently metamodeling the core and each extension in a separate package, to then assemble them together. So for example, when we applied this approach to the case of Flora, we developed one separate metamodel package for (1) *First-order logical terms* that are common to Prolog and many other logic and rule-based languages, (2) *Prolog programs and clauses* that reuses the logical terms package, and (3) each *orthogonal extension* of either logical terms or Prolog clauses used in CTFL programs. From such minimal

metamodel units, larger metamodels for HL, STL, CTL, FL, SFTL *(i.e.,* for Flora) and CTFL can then be obtained through assembly. This compositional approach brings five clear benefits. The first is *cognitive complexity management* for such large meta-modeling tasks. For example, the fully assembled Flora metamodel contains over 275 elements. Hence, much simplicity was gained by decomposing it into nine packages. The second benefit is the creation of valuable course *didactic assets* that visually contrast basic language concepts and platform services and clarify the many different ways in which they can be integrated. For example, the compositional Flora metamodel is such an asset for teaching the distinct principles and complementary strengths of logic, object-oriented and imperative language concepts for programming, knowledge representation and data manipulation. The third benefit is the *reusability of minimal metamodel units* for other languages or platforms. For example, we have already reused the first-order logical term package of the Flora metamodel to build a metamodel of the language *Constraint Handling Rules* [7]. The fourth benefit is to provide a sound basis for *representing integration semantics* in UML and OCL which are more accessible than the mathematical notations generally used for such task. In the case of Flora, while each of its sub-language possesses denotational and operational semantics, the unification of theses into a single framework is still incomplete; metamodeling the semantics of each sub-language should bring valuable insights towards such unification. The fifth benefit of compositional metamodeling is compiler MDD based on model transformations. For example the Flora to XSB compiler could be specified as a set of declarative QVT relations [14] between elements of the STFL and HiLog metamodels.

These benefits are pervasive since many modern, powerful languages result, like Flora does, from successive and partially orthogonal extensions from an initial simple core. For example, the semantic web language standards put forward by the World Wide Web consortium (W3C) also evolved this way: an initial core, RDF [1] was successively extended to yield RDFS, DAML, DAML-OIL, and finally OWL [16]. In addition, all these languages reuse the syntactic core XML syntax, and DAML-OIL itself resulted from the integration of DAML with OIL.

3 Illustrative Case-Study: HiLog as an Extension of Prolog

While we assembled the minimal unit metamodel packages into the whole Flora metamodel, we were confronted several times with the need to reuse metamodel elements from one package while generalizing them in the package assembly. We now explain why such reuse need cannot be addressed by the facilities currently provided by the MOF 2.0 and UML 2.0 standards by focusing on a *single* assembly step: the one that takes as input the Prolog metamodel and a metamodel of high-order predicates and that results in the HiLog metamodel. We first present (simplified) non-compositional versions of the Prolog and HiLog metamodels and explain their main concepts. We then further zoom on only the relevant elements in these metamodels necessary to illustrate the *reuse and generalize* need, and successively show that elements from the Prolog metamodel for first-order predicates *cannot* be reused and generalized to define high-order predicates in the HiLog metamodel by using either

`import`, `merge` or `combine`. We then specify a new `reuse and generalize` package facility and illustrate its use for such case.

3.1 A Non-compositional Simplified Metamodel of Prolog

The non-compositional, simplified Prolog metamodel is shown in Fig. 1. It shows that a Prolog program is a set of clauses, with each clause consisting of a premise that is a Prolog query and a single conclusion that is a first-order logic atom. A Prolog query is a tree of arbitrary depth which leaves are first-order logic atoms and which non-leaf nodes are one of the two logical connectives `and` or `or`. A first-order logic atom is also a tree of arbitrary depth which leaves are either constant symbols or first-order variables, and which non-leaf nodes can only be constant symbols. Each sub-tree is a called a logical term, which root is called the functor of the term and which depth one sub-trees are called its arguments. Non-functional terms are depth zero sub-trees and opposed to functional terms which depth is at least one. A ground term is a sub-tree of arbitrary depth that is free of variables.

3.2 A Non-compositional, Simplified Metamodel of HiLog

High-Order Logic (HiLog) extends Prolog with high-order syntactic sugar while semantically remaining first-order. At the program, clause and query levels, HiLog follows exactly the same construction rules as Prolog. The former extends the latter only at the lower logical atom level. The extension is twofold: (1) HiLog, allows programs in addition to terms as arguments, and (2) HiLog allows arbitrary terms (functional or not, ground or not) as functors instead of restricting them to constant symbols (i.e., non-functional ground terms) as Prolog does.

For example, P(f(X))(G(Y), (G, c :- X, p(Y(P)))). is a valid HiLog term but not a Prolog term for three reasons: (1) its functor is a compound term P(f(X)), (2) its first argument's functor is a variable G and (3) its second argument is a program made of two clauses, G. and c :- X, p(Y(P)). HiLog extends Prolog with meta-programming, meta-reasoning and metadata definition and query facilities within the logical paradigm under well-defined first-order declarative semantics. It brings to logic programming the high-order syntax that is key to the versatility of functional programming. In the HiLog metamodel of Fig. 2 the first of the two ways in which HiLog extends Prolog is reflected by the fact that the `functor` meta-association outgoing from the `FunctionalTerm` meta-class targets the `Term` meta-class, instead of the `Constant` meta-class as in the Prolog metamodel. The second extension is reflected by the introduction of the new meta-class `LogicalArgument` as the target of the `arg` meta-association outgoing from the `LogicalAtom` meta-class. This new `LogicalArgument` meta-class generalizes the two meta-classes `FunctionalTerm` and `Program`.

3.3 Reusing and Generalizing Metamodel Elements from Prolog in HiLog

The relation between HiLog and Prolog can be summarized as follows: “A HiLog program is a Prolog program, *except that,* (a) the functor of the atoms of its clauses can recursively be compound and/or non-ground terms and (b) the arguments of the atoms of its clauses can recursively be HiLog programs.” As a language, HiLog thus

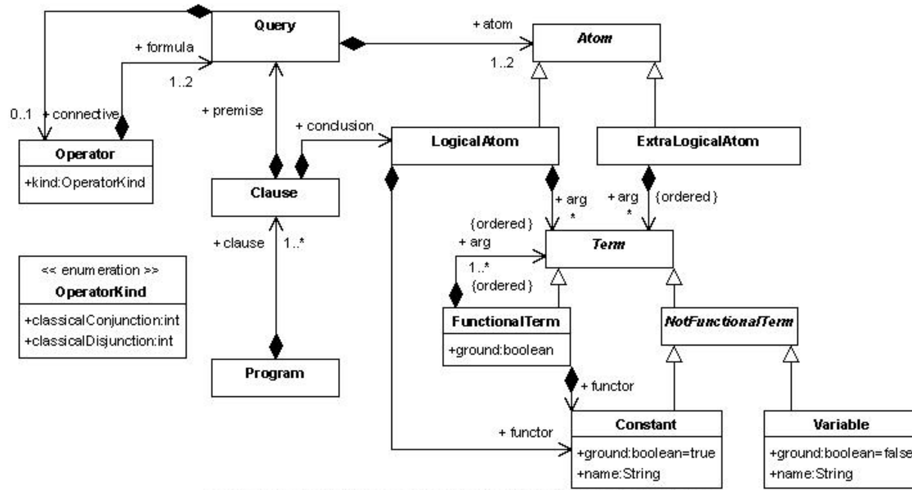

Fig. 1. Non-compositional, simplified Prolog metamodel

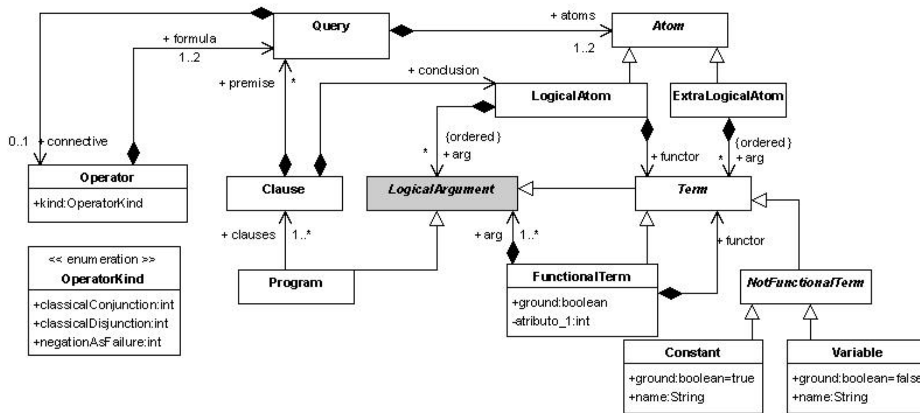

Fig. 2.Non-compositional simplified HiLog metamodel.

reuses Prolog in three ways. First it reuses its lexical categories `constant` and `variable`. Second it reuses its functional roles `functor`, `argument` and `connective`. Third it reuses its syntactic rules to build clauses from terms and connectives and to build programs from clauses. However, HiLog reuses Prolog by *generalizing* it: the HiLog syntactic rules to build terms from constants, variables and programs fulfilling the functor and argument roles of a term are less restrictive than the corresponding rules in Prolog. Thus, every Prolog term (respectively clause and program) is also a HiLog term (respectively clause and program), but the converse is false. Focusing for the sake of clarity at the term level, this *reuse and generalize* relation between HiLog and Prolog is summarized in Fig. 3. What we want to precisely capture is the following:

1. Each element `E` (meta-class or meta-association) in the Prolog metamodel package that represents a construct that is exactly the same in Prolog and HiLog shall be available from the HiLog metamodel without need for redefinition;
2. Each element `E` (meta-class or meta-association) defined in the HiLog metamodel package that was already defined in the Prolog package is a new element `HiLog::E` that generalizes `Prolog::E`.
3. A new element `G` (meta-class or meta-association) defined in the Hilog metamodel package can generalize an element `S` (meta-class or meta-association) already defined in the Prolog package, i.e, `HiLog::G` can generalize `Prolog::S`.

So in the example of Fig. 3, we want:

- The `Term`, `NonFunctionalTerm`, `Constant` and `Program`[3] meta-classes of the Prolog package to be reusable “as is” by elements of the HiLog package;
- The `LogicalAtom` meta-class and the `arg` and `functor` meta-associations of the Prolog package to be become specializations of the new `LogicalAtom` meta-class and the `arg` and `functor` meta-associations of the HiLog package (respectively).
- The `LogicalArgument` meta-class can generalize the meta-classes `Term` and `Program` already defined in Prolog package.

In the following subsections, we examine whether any of the three metamodel package composition facilities provided by MOF 2.0 can capture such relation.

3.4 Should the HiLog Metamodel `Import` the Prolog Metamodel?

In the MOF standard [11] the package `import` mechanism is defined as “a relationship that allows the use of unqualified names to refer to package members from other namespaces”. It is a one-way relationship: when a package P imports a

[3] Omitted from Fig. 3 but linked to `LogicalAtom` through association navigation in Fig. 1 and Fig. 2.

package Q, the elements of P can be linked to the elements of Q but not vice-versa. In our case, since we want to reuse Prolog package elements in the HiLog package, the only possible direction is to `import` the Prolog package from the HiLog package. In that direction, `import` fulfills the *reuse* part of our *reuse and generalize* need. However it then also prevents the fulfillment of the *generalize* part. This is illustrated in Fig. 4. The `Term` meta-class of the Prolog package *cannot* be linked as needed to the `LogicalArgument` of the HiLog package meta-class by a specialization association, for it would break the unidirectionality of the `import` dependency between the two packages.

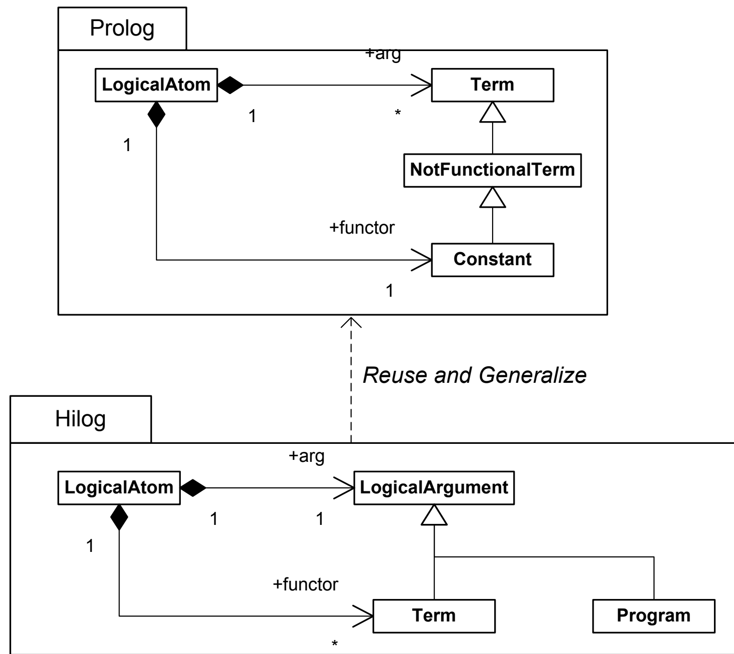

Fig. 3. Reuse and Generalize relationship between HiLog and Prolog

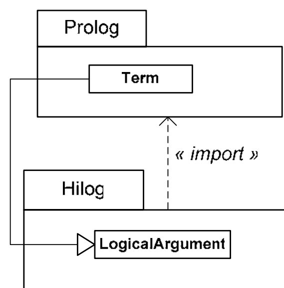

Fig. 4. `Import` cannot stand for reuse and generalize.

3.5 Should the Prolog Metamodel Be Merged Within the HiLog Metamodel?

The merge mechanism has been first defined in the UML2.0 Infrastructure [12]. Merging a package Q within a package P can be understood as an "alias" equivalent to the following action sequence: (1) for each element (meta-class or meta-association) E of the merged package Q, create a copy E' of E in the merging package P; (2) Import Q from P; and (3) for each copied element E, create an inheritance association that states that its copy E' specializes E.

In our case, since we want to reuse the Prolog package in the HiLog package, our only option is to merge the Prolog package within the HiLog package. The result of doing so is illustrated in Fig. 5. Let us now examine whether it fulfills the two requisites for our *reuse and generalize* need. Just as `import` (that `merge` in effect extends), `merge` fulfills the *reuse* part but it fails to fulfill the *generalize* part. For example, in the merged metamodel, the HiLog::Term meta-class *specializes* the Prolog::Term meta-class. But in fact it should *generalize* it since everywhere a HiLog term can appear in a HiLog program, a Prolog term is also valid, but some HiLog terms are not valid in some places where a Prolog term can appear in a Prolog program. In fact, merge can only be used to fulfill a *reuse and specialize* need, but not a *reuse and generalize* need as in the case of Prolog and HiLog.

3.6 Should the Prolog Metamodel Be `Combined` with the HiLog Metamodel?

As the `merge` relationship defined in the UML2.0 Infrastructure brings a lot of inheritance between elements of the merging and merged packages, MOF2.0 defined a variant relationship, historically named `combine` [11]. This relationship is also defined in the latest UML2.0 Superstructure [13], under a different name: `package merged`. To distinguish it more clearly from `merge`, we will use its historical `combined` name in the rest of the paper. In the previous section, we saw that `merge` is a complex operation that can be decomposed in three basic steps: (a) copy, (b) import and (c) inherit. `Combine` differs from `merge` in that it only performs the first "copy" step. At first, it seems adequate to fulfill our *reuse and generalize* need, since we saw that with `merge,` the *reuse* part was fulfilled by the "copy" step but the *generalize* part was made impossible by the "inherit" step that was in the wrong direction. However, consider the situation where a meta-association A occurs both (a) between meta-classes C and D in the package Q to reuse and (b) between C and a generalization G of D in receiving package P. This a standard situation when using combine to fulfill a *reuse and generalize* need. It is illustrated in Fig. 6. where the same meta-association `arg` links the meta-class `LogicalAtom` to the meta-class `Term` in the Prolog package to reuse, while it links the same meta-class `LogicalAtom` to the meta-class `LogicalArgument` that generalizes `Term` in the receiving HiLog package. In such cases, the package resulting from combining Q with P *is not a valid* MOF 2.0 metamodel since it includes two meta-associations *with the same name* A that links the same meta-class C to *two distinct* meta-classes G and D. This is illustrated in Fig. 7 that shows the package resulting from combining the Prolog and HiLog packages. In this example, two copies of the `arg` meta-association link the meta-class `LogicalAtom` to two distinct meta-classes,

LogicalArgument and Term, and two copies of the functor meta-association also link of LogicalAtom to both the Term and Constant meta-classes. The resulting metamodel is thus invalid.

3.7 Our Proposal: A New Metamodel Assembly Facility

Since none of the three metamodel package assembly facilities currently provided either MOF 2.0 or UML 2.0 can satisfactorily fulfill a pervasive need for compositional metamodeling, we propose a fourth one that we call reuse and generalize. It is based on combine but corrects the flaw that we identified in the latter for reusing while generalizing elements of a metamodel package Q into another package P. This facility creates a new resulting package that assembles elements from P and Q by the following action sequence:

1. For each element (meta-class or meta-association) E appearing in either packages Q and P, create a copy E' of E in the resulting package R.
2. Whenever this results in conflicting pairs of meta-associations with the same name, one linking a meta-class C to a meta-class G and another linking a meta-class C to specialization D of G, delete the latter.

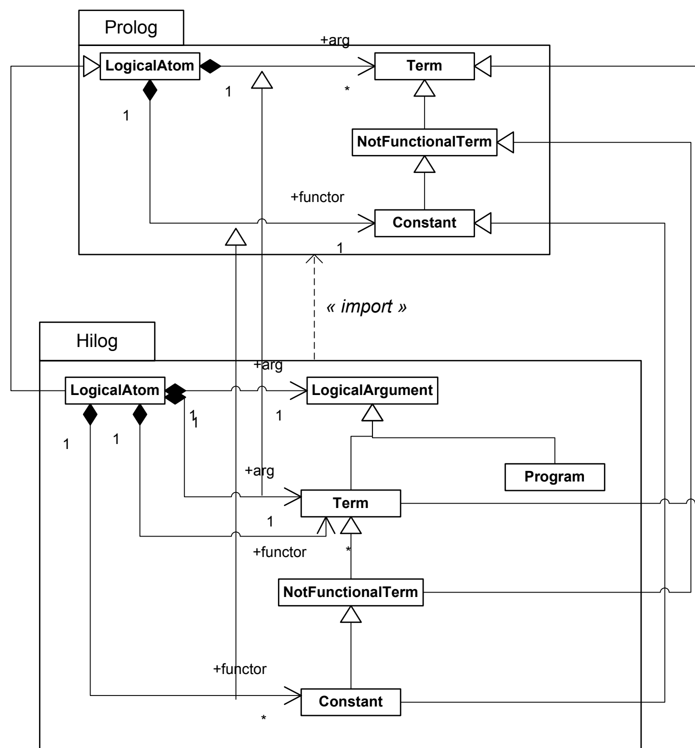

Fig. 5. Merge cannot stand for reuse and generalize.

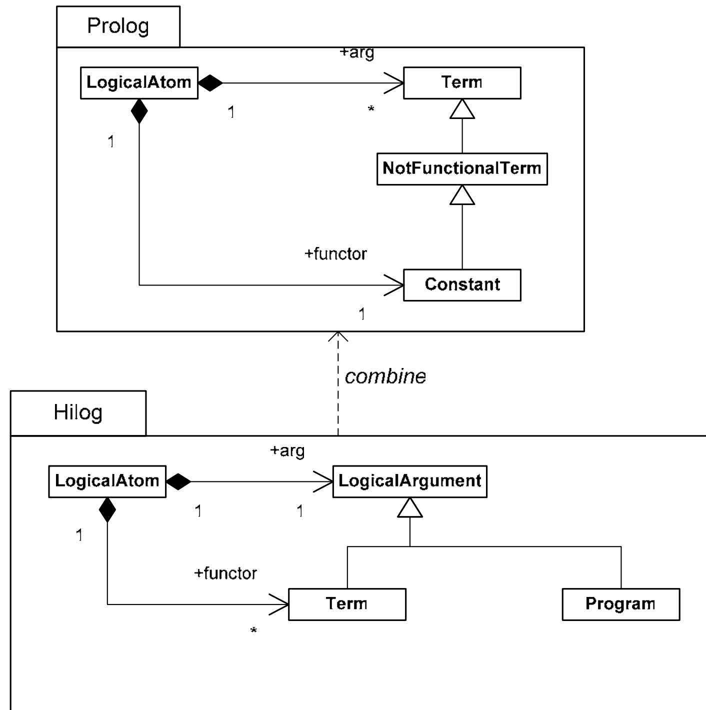

Fig. 6. Input packages of `combine` and of `reuse and generalize`.

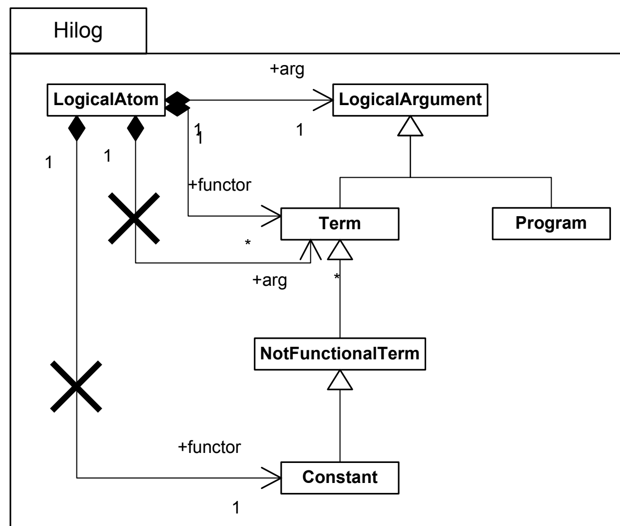

Fig. 7. Resulting package of `combine` and of `reuse and generalize`.

The result of `reuse and generalize` the Prolog package in the HiLog package in shown in the same Fig. 7, together with the result of `combined`. The difference is the crossed-over links, present with `combined` but absent with `reuse and generalize`. The latter thus contains only a single `arg` meta-association that links the `LogicalAtom` meta-class to the `LogicalArgument` meta-class and a single `functor` meta-association that links `LogicalAtom` to the `Term` meta-class. It is thus a valid MOF 2.0 metamodel that reuses the Prolog metamodel while generalizing it elements in the resulting HiLog metamodel. Note that the HiLog *package* that forms the second input to this `reuse and generalize` transformation is *not* a HiLog *metamodel* for it captures only the constructs *proper* to HiLog that define how it extends Prolog. The current version of the whole Flora metamodel was assembled from nine packages and sub-packages linked together by two instances of `import` and seven instances of `reuse and generalize`.

4 Related Work

Using a subset of the UML structural infrastructure similar to MOF, Clark et al. [5] proposed two new package assembly facilities. The first that they call "merge", but that is distinct from the `merge` of MOF, addresses the case of the same meta-class occurring in the two packages to assemble. In the "merging" package, the meta-attributes and meta-associations of such meta-class becomes the union of those in its occurrences in the two "merged" packages. The second facility, called *renaming* allows equating the names of two elements with two distinct names in the two packages prior to a "merge". Later, the same authors [6] proposed a Metamodeling Framework using a language that is distinct from MOF and that allows the definition of parametric model elements templates. This framework includes package *specialization* assembly facilities that deal with such templates.

Addressing the same issue, Ledeczi [9] proposed three assembly facilities: one that is much similar to Clark et al.'s "merge", one that restricts the union of the meta-class elements in the "merging" package to its attributes and containment associations in the "merged" package, and one that restricts it to its complementary non-containment associations.

Mens et al. [10] proposed package assembly facilities for collaborative diagrams. Since such diagrams are not used at the metamodeling level, these facilities do not seem to be easily applicable to metamodel assembly issues.

In short, none of the facilities proposed in these works addresses the specific *reuse and generalize* need that we identified.

5 Conclusion

Many powerful computational languages and platforms result from a historical process of gradually extending an initial core with largely orthogonal and complementary constructs inspired from other languages. There are great benefits to metamodeling such languages and platforms in a compositional way that reflects this

historical maturation and clearly separates concerns. When doing so, one immediately feels the need for a metamodel package assembly facility that allows both *reusing* the elements of two basic packages, each one focused on a single concern, and *generalizing* them in a resulting package that captures their integration. In this paper, we have shown that none of the three metamodel assembly facilities currently provided by MOF and UML fulfills such need:

- `Import` because it prohibits imported elements to specialize elements of the importing package;
- `Merge` because it implicitly makes the reused elements generalizations instead of specializations of the new ones;
- `Combine` because when a reused meta-class is generalized, its meta-associations are not generalized, but instead *duplicated* at the level of its generalization, resulting in an invalid model.

We thus proposed a new `reuse and generalize` facility that is inspired from `combine` but that correctly generalizes instead of duplicating meta-associations in such cases. The ever widening scope of MDD is likely to reveal many other metamodel reuse needs beyond the "reuse as is" and "reuse and specialize" currently provided by MOF and the "reuse and generalize" addressed in this paper. In future work, we intend to create a catalog of metamodel reuse needs and to identify how these needs can be addressed by a minimal set of primitive reuse operators together with an algebra that defines semantically sound complex compositions of such operators.

Acknowledgements

The research presented in this paper was sponsored by research grant 371/01 from CAPES-COFECUB, by one doctoral research fellowship from CNPq, by the ModelWare project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006), and by the "Model Driven Development Integration" (MDDi) project from the Eclipse Foundation. We would like to thank Marie-Pierre Gervais for her insightful feedback on a preliminary version of this paper.

References

[1] Birbeck, M., Ozu, N. et al.: Professional XML. 2nd Ed. Wrox (2001)
[2] Bonner, A. and Kifer, M.: Transaction Logic Programming. Technical Report CSRI-323. Computer Systems Research Institute, University of Toronto (1995)
[3] Bonner, A. and Kifer, M.: Concurrency and Communication in Transaction Logic. Joint International Conference and Symposium on Logic Programming. Bonn, MIT Press (1996).
[4] Chen, W., Kifer, M. and Warren, D.S.: HiLog: A Foundation for High-Order Logic Programming. Journal of Logic Programming. 15(3) (1993) 187-230
[5] Clark, T., Evans, A. and Kent, S.: A Metamodel for Package Extension with Renaming. International Conference on the Unified Modeling Language (2002) 305-320

[6] Clark, T., Evans, A. and Kent, S.: Engineering Modelling Languages: A Precise Metamodeling Approach. Fundamental Approaches to Software Engineering (FASE) International Conference. Lecture Notes in Computer Science, Vol. 2306. Springer-Verlag (2002) 159-173
[7] Frühwirth, T. and Abdennadher, S.: Essentials of Constraint Programming. Series: Cognitive Technologies.Springer. (2003)
[8] Kifer, M., Lausen, G. and Wu, J.: Logical Foundations of Object-Oriented and Frame-Based Languages. Journal of the ACM 42(4). (1995) 741-843.
[9] Ledeczi A, Nordstrom, G., Karsai, G., Volgyesi, P. And Maroti, M.: On Metamodel Composition. Conference Control Applications, IEEE Press. Mexico City, Mexico (2001) 84-90
[10] Mens, T., Lucas, C. and Steyart, P.: Supporting Disciplined Reuse and Evolution of UML Models. PSMT – Workshop on Precise Semantics for Software Modeling Techniques in UML Conference. (1998) 378-392
[11] OMG.: The MOF 2.0 specification. http://www.omg.org/mof (2003)
[12] OMG.: The UML 2.0 Infrastructure specification. http://www.omg.org/uml (2003)
[13] OMG.: The UML 2.0 Superstructure specification. http://www.omg.org/uml (2003)
[14] The QVT-Merge Group. QVT 1.8.: Revised submission for OMG MOF 2.0 Query/Views/Transformations Request For Proposal. 2004.
[15] Ramalho, F., Robin, J. and Schiel, U.: Concurrent Transaction Frame Logic Formal Semantics for UML Activity and Class Diagrams. Electronic Notes in Theoretical Computer Science, 95(17). (2004)
[16] The World-Wide Web Consortium. Web Ontology Language. http://www.w3.org/2004/OWL. 2004 (2004)
[17] Yang, G.: A Model Theory for Nonmonotonic Multiple Value and Code Inheritance in Object-Oriented Knowledge Bases. PhD. Thesis, Computer Science Department, Stony Brook University of New York. (2002)
[18] Yang, G., Kifer, M. and Zhao, C. FLORA-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web. 2nd International Conference on Ontologies, Databases and Applications of Semantics (ODBASE), Catania, Italy. (2003) 671-688.