Evolution of Modelling Languages

Bart Meyers
Modelling, Simulation and Design Lab (MSDL)
University of Antwerp
Middelheimlaan 1
B-2020 Antwerp, Belgium
Bart.Meyers@ua.ac.be

Hans Vangheluwe
Modelling, Simulation and Design Lab (MSDL)
University of Antwerp
Middelheimlaan 1
B-2020 Antwerp, Belgium
Hans.Vangheluwe@ua.ac.be

ABSTRACT

Over the course of the complete life cycle of complex software-intensive systems and more importantly of entire product families, evolution is inevitable. Not only instance models, but also entire modelling languages are subject to change. This is in particular true for domain-specific languages. Up to this day, modelling languages are evolved manually, with tedious and error-prone migration of for example instance models as result. This position paper discusses the different evolution scenarios for various kinds of modelling artifacts, such as instance models, meta-models and transformation models. Subsequently, evolution is de-composed into four primitive scenarios such that all possible evolutions can be covered. The pre-requisites for implementing this approach are discussed, showing how a number of these are not yet supported by Fujaba. We suggest that using our structured approach in Fujaba will allow a relatively straightforward implementation of (semi-)automatic model evolution.

1. INTRODUCTION

In software engineering, the evolution of software artifacts is ubiquitous. These artifacts can be programs, data, requirements, documentation, but also languages. Language evolution applies in particular to domain-specific modelling (DSM), where domain-specific languages (DSLs) are specifically designed to minimize accidental complexity by using constructs closely coupled with their domain. This results in a reported productivity increase of a factor 5 to 10 [10]. DSLs must be quickly built and used, and grow incrementally. A formal underpinning for DSM is given by multi-paradigm modelling (MPM) [15].

The high dependence on their domains and the need for instant deployment make DSLs highly susceptible to change. Such an evolution of a language can have substantial consequences, which will be explained throughout this paper. Early adopters of the model-driven engineering paradigm dealth with this evolution problem manually. However, such a pragmatic approach is tedious and error-prone. Without proper methods, techniques and tools to support evolution, model-driven engineering in general and domain-specific modelling specifically will not scale to industrial use.

$$\begin{tikzcd}
 & & MM & & MM_{T_i} \\
MM_{\kappa_i} & & m \arrow[u, dotted] \arrow[rr, "T_i"] \arrow[dll, <->, "\kappa_i"'] \arrow[ddr, "M"] & & T_i(m) \arrow[u, dotted] \\
\kappa_i(m) \arrow[u, dotted] & & & MM_M & \\
 & & & M(m) \arrow[u, dotted] &
\end{tikzcd}$$

Figure 1: A model and its relations in MPM.

1.1 Modelling Languages

To allow for a precise discussion of language evolution, we briefly introduce the concepts fundamental to modelling languages, in the context of multi-paradigm modelling [5].

The two main aspects of a model are its *syntax* (how it is represented) and its *semantics* (what it means).

Firstly, the syntax comprises *concrete syntax* and *abstract syntax*. The concrete syntax describes how the model is represented (in 2D vector graphical form for example), which can be used for model input as well as visualization. The abstract syntax contains the essence of the structure of the model (as an abstract syntax graph), which can be used as a basis for semantic anchoring. A single abstract syntax may be represented by multiple concrete syntaxes. There exists a mapping between a concrete syntax and its abstract syntax, called the *parsing mapping function*. There is also an inverse mapping, called the *pretty printing mapping function*. Mappings are usually implemented, or can be at least represented, as model transformations.

Secondly, the semantics of a model are defined by a complete, total and unique *semantic mapping function* which maps every model in a language onto an element in a *semantic domain*, such as differential equations, Petri Nets, or the set of all behaviour traces. Semantic mapping functions are performed on the abstract syntax for convenience.

A meta-model is the finite and explicit description of the abstract syntax of a language. Often, the concrete syntax is also described by (another) meta-model. Semantics are however not covered by the meta-model. The abstract syntax of the semantic domain itself will of course conform to a meta-model in its own right.

Figure 1 shows the different kinds of relations a model *m* is involved in. Relations are visualized by arrows, "conform to"-relationships are dotted arrows. The abstract syntax model *m* conforms to its meta-model *MM*. There is a bidirectional relationship κ_i (parsing mapping function and pretty printing mapping function) between *m* and a concrete syntax $\kappa_i(m)$. $\kappa_i(m)$ conforms to its meta-model MM_{κ_i}. Semantics are described by the semantic mapping function *M*, and map *m* to a model *M(m)*. *M(m)* has syntax which conforms to MM_M. Additionally, there may be other transformations T_i defined for *m*.

2. RELATED WORK

In this section other work related to evolution is presented and some useful concepts are introduced.

2.1 Model Differencing

In order to be able to model evolution in-the-large, one should be able to model differences between two versions of a model. This can of course be done by using lexical differencing, as used for text files, on the data representation of the model. However, the result of such analysis is often not useful, as (1) the differences occur at the granularity level of nodes, links, labels and attributes and (2) models are usually not sequential in nature and equivalences between models will not be taken into account. Hence, model differencing should be done at the appropriate level of abstraction. Some useful research has been done in this area [1, 17, 13, 23, 4]. Existing approaches typically rely on the abstract syntax graphs (ASGs) of the two models, and mainly traverse both graphs in parallel. Nodes in the graphs are matched by matching unique identifiers [1, 17], or by a number of heuristics [13, 23]. However, no large-scale version control system that computes the differences between graph-like models exists yet.

Next to the problem of finding differences, one should be able to *represent* them as a model, which we will call the *delta model*. There are two kinds of representations: operational and structural representations. In the operational representation, the difference between two versions of a model is modelled as the edit operations (create/read/update/delete) that were performed on the on one model to arrive at the other [1, 8]. When these operations are recorded live from a tool, this strategy is quite easy and powerful, but dependent on that particular tool and hard to visualize. In structural representations, either the model (or its DOM representation) is coloured [17, 23, 13, 19] or a designated delta model is created which can be used by modelling tools as yet another model [4, 20].

2.2 Model Co-Evolution

When the syntax of a modelling language evolves (i.e., the meta-model evolves), the most obvious side-effect is that its instance models are not conform to the new meta-model. Therefore, the co-evolution of models has become a popular research topic. This research is inspired by evolution in other domains, such as grammar evolution [18], database schema evolution [2] and format evolution [12].

It is widely accepted that a model co-evolution (i.e., migration) is best modelled as a model transformation [24, 11, 20, 9, 7, 22, 21, 3, 8], which we will call the *migration transformation*. Grushko et al. write this transformation manually using the Epsilon Transformation Language (ETL) [7].

Most of the approaches however define some specific operations as building blocks for evolution, similar to the operational representation of model differences. Such operations typically include "create meta-class", "restrict multiplicity on meta-association" or "rename meta-attribute" and are related to object-oriented refactoring patterns. These operations, which we will call *delta operations*, are reusable. Conveniently, migration transformations can be generated from sequences of delta operations. It is important that any possible evolution can be modelled, but there is a general consensus that the proposed sets of delta operations do not suffice. In a very recent approach, Herrmannsdörfer et al. try to solve this problem by repeatedly extending their list of delta operations [8]. In addition, they support customized evolution. This ensures expressiveness, but the migration transformation code must be implemented manually.

Table 1: Evolution operations as presented in [3].

Operation type	Operation
Non-breaking operations	Generalize meta-property Add (non-obligatory) meta-class Add (non-obligatory) meta-property
Breaking and resolvable operations	Extract (abstract) superclass Eliminate meta-class Eliminate meta-property Push meta-property Flatten hierarchy Rename meta-element Move meta-property Extract/inline meta-class
Breaking and unresolvable operations	Add obligatory metaclass Add obligatory metaproperty Pull metaproperty Restrict metaproperty Extract (non-abstract) superclass

Gruschko et al. make a distinction between non-breaking, resolvable and unresolvable operations. Non-breaking operations do not require co-evolution. Inconsistencies caused by resolvable operations can be resolved by co-evolution. However, model co-evolution for unresolvable operations requires additional information in order to execute. For example, when a "create obligated meta-feature"-operation is performed on a meta-model, then a new feature is created for each instance. However, the information about what the initial value of this feature will be, is unknown, as it differs from model instance to model instance. As an illustration, the operations proposed by Cicchetti et al. [3] are shown in Table 1. Note the similarities with refactoring patterns.

3. EVOLUTION FOR MPM

While model co-evolution as described above implements automation to some extent, there are other artifacts that might have to co-evolve. This section presents an exhaustive survey of possible evolutions and co-evolutions.

3.1 Syntactic Evolution

To get a general idea of the consequences of evolution, let us go back to Figure 1. When *MM* evolves, all models *m* have to co-evolve, which was discussed in Section 2.2. However, as the relations of Figure 1 suggest, the evolution of *MM* might affect other artifacts. First, similar to *m*, (the domain and/or image of) transformations such as κ_i, T_i and *M* might no longer conform to the new version of the metamodel. As a consequence, they too have to co-evolve. This makes all relations (syntactically) valid once again, which means that the system is syntactically *consistent* again. In short, meta-model evolutions can only be useful when both their model instances and related transformation models can co-evolve.

However, there are more scenarios. Firstly, it is possible that the meta-model changes in such a way that the co-evolved models become structurally different, for example by removing a language construct. This means that each transformation defined for each co-evolved model has to be re-executed. The resulting co-evolved models can also be structurally different, so a chain of required evolution transformation executions may be required.

Secondly, changes made to one meta-model can reflect on another meta-model. For example, when a meta-element is added to a meta-model, a new meta-element is often also added to the meta-model of the concrete syntax(es) in order to be able to visualize this new construct. A similar effect can occur between any two related (by transformation) meta-models. In this sense, a chain of meta-model changes is again possible.

Thirdly, until now, we only discussed meta-model evolution as the driving force. Evolution of other artifacts, such as instance models and transformation models should also be taken into account. The case of the evolution of a model is trivial: related models can co-evolve by executing the respective transformations. Note however that a co-evolved model may be a meta-model, so that might trigger a number of co-evolutions of its own.

The case of the evolution of a transformation model can get complicated. In many cases though, the evolved transformation simply has to be executed again on each model it is defined for. However, this would restrict a transformation evolution to remain compliant to its source and target metamodels, which is not always what we want. For example, it might be possible that a new language is created by mapping rules for each language construct of an existing language. This is in particular convenient for creating a concrete syntax. On top of this, there are two additional special cases of transformation evolution. Firstly, the evolution of the parsing mapping function or the pretty printing mapping function requires the other one to co-evolve in order to maintain a meaningful relation between abstract and concrete syntax. Such a co-evolution can be generalized to any bidirectional transformation. Secondly, the evolution of the semantic mapping function requires a means to reason about semantics in order to trigger co-evolution, which brings us to the concept of semantic evolution.

3.2 Semantic Evolution

As mentioned above, semantics of a model are defined by its semantic mapping function to a semantic domain. Some analysis can be performed on models in this semantic domain (for example: check for a deadlock in a Petri Net). The results of this analysis can be considered a *property* of the model, or *P(m)*. A semantic mapping function is constructed in such a way that some properties $P_M(m)$ hold both for a model and for its image under the semantic mapping (i.e., the intersection of both property sets). These common properties have to be maintained throughout evolution. An evolution is a semantic evolution if some of these properties change. This typically happens when the requirements of a system change.

In general, when a model *m* in a formalism whose semantics is given by semantic mapping function *M* evolves to *m'*, then $P_M(m')$ must be exactly $P_M(M(m))$ modulo the intended semantic changes. In general, when two versions of a system are (a) equal modulo their intended syntactic and semantic changes and (b) syntactically consistent, then the evolution of the system is *continuous*. Only continuous evolutions are deemed correct (and meaningful).

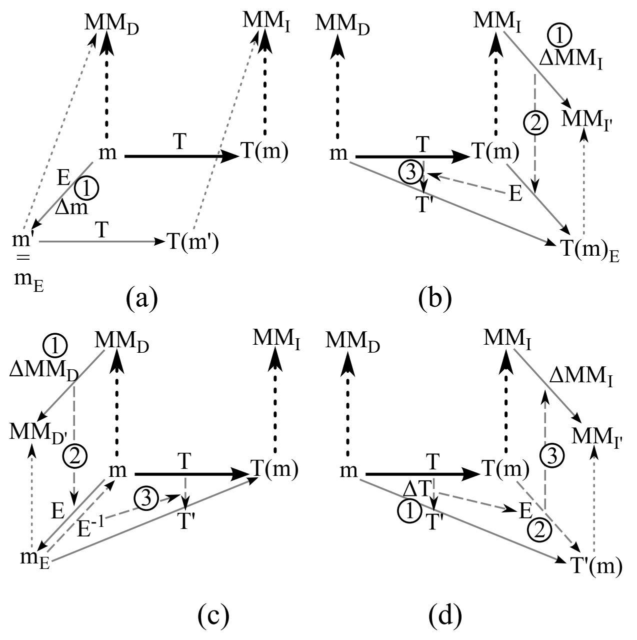

Figure 2: Co-evolution in (a) model evolution, (b) image evolution, (c) domain evolution and (d) transformation evolution.

4. DE-CONSTRUCTING EVOLUTION

As discussed in the previous section, there are infinitely many possible co-evolution scenarios. Nevertheless, these scenarios can always be broken down into a few basic ones. Figure 2 shows the possibilities. Again, arrows are transformations and dotted arrows are "conforms to"-relationships. Dashed arrows denote a (semi-)automatic generation. Each diagram starts from a bold relation between two meta-models MM_D and MM_I, modelled as a transformation *T* of models *m*.

4.1 Model Evolution

Figure 2 (a) shows model evolution. Some model *m* evolves to *m'*. In step 1 (the only step), a delta model $\Delta m$is constructed (either automatically or manually) that models the evolution of *m* to *m'*. This means that $m' = m + \Delta m$. The evolution itself is typically represented as a migration transformation, namely *E*. The equation $m_E = m + \Delta m = m'$ is valid. As previously discussed, because *m* evolved to *m'*, every transformation *T* must be executed again, resulting in *T(m')*, conform to MM_i.

4.2 Image Evolution

Image evolution is shown in Figure 2 (b). Suppose that a meta-model MM_I evolves to $MM_{I'}$. In step 1 a delta model ΔMM_I is constructed to represent the difference between MM_I and $MM_{I'}$. In step 2 a migration transformation *E* is generated out of ΔMM_I. The execution of *E* co-evolves models $T(m)$ to $T(m)_E$, so that they conform to the new meta-model $MM_{I'}$. Moreover, the execution transformation *T* has to result in valid models (i.e., conform to $MM_{I'}$). As a consequence, *T* has to co-evolve to a new transformation *T'* (as in step 3), which is able to transform every possible *m* that conforms to MM_D, to $T(m)_E$. The diagram presents a solution for the

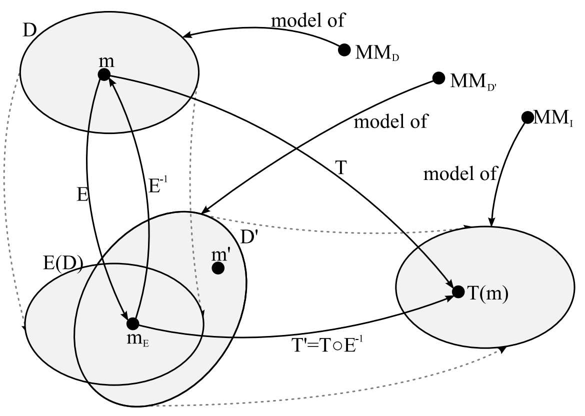

Figure 3: Set representation of domain-evolution. The evolution *E(D)* does not map onto *D'* exactly. For *m'*, the constraint *T' = T*$\circ$*E*$^{-1}$ does not hold!

generation of this *T'*: for every *m*, *T'(m) = E(T(m))* holds, or in short, *T' = E* $\circ$ *T*. The co-evolution *T'* can be simply composed out of *T* and *E*.

4.3 Domain Evolution

Figure 2 (c) shows domain evolution, where MM_D evolves. The artifacts that co-evolve are similar to image evolution. This time however, *T* can be expressed as $T' = T \circ E^{-1}$. So, in this case, an inverse transformation E^{-1} needs to be constructed. Unfortunately, this equation does not hold for the entire domain *D'*, as shown in Figure 3. The migration transformation *E* projects the entire domain *D* to *E(D)*, but it is possible that $E(D) \neq D'$. For *m* in Figure 3 it may be possible possible to construct E^{-1} such that $T'(m_E) = T(E^{-1}(m_E))$ holds. However, for *m'*, which is an element of $D' \setminus E(D)$, this is not possible. Nevertheless, *T'* must apply to its entire domain *D'*, so the equation $T' = T \circ E^{-1}$ can not be used for all possible models conform to $MM_{D'}$.

4.4 Transformation Evolution

Figure 2 (d) shows transformation evolution. The requirements of a system can change, resulting in the adjustment of the (desired) properties of a model. If transformations evolve according to a delta model ΔT, it is possible that they only have to be executed once again. In this case, the changes on the transformation are limited: the image of *T'* must conform to MM_i. As previously discussed, other artifacts might possible co-evolve. In this case, a migration transformation *E* must be composed from which a delta model ΔMM_i can be constructed.

4.5 Evolution Scenario Amalgamation

Using a combination of these four scenarios, all possible evolutions can be carried out. Note however that the problem of Figure 3 applies, so automated co-evolution is not always possible. The so-called unresolvable changes can be classified as models in $E(D) \setminus D'$. On the other hand, the transformation has to support the models in $D' \setminus E(D)$. We call this the *projection problem*. In general, the projection problem arises when $dom_E(T) \not\subseteq dom(T)$.

5. PRE-REQUISITES FOR EVOLUTION

Following the discussion above, the proposed approach depends on a few more general techniques. Many of these pre-requisites are currently not supported by Fujaba. As a consequence, implementing all forms of evolution is currently not feasible in practice in Fujaba. The following pre-requisites (in order of priority) are necessary or at least useful for implementing evolution:

- *higher order transformation*: the automatic generation of migration transformations out of delta models requires support for higher order transformations, which are transformations that take other transformations as input and/or output. This is not supported by Fujaba, as the transformation language is not modelled explicitly (i.e., the meta-model is not available). There are several other uses for higher order transformation, in and out of the context of evolution, which are not discussed here [3, 16, 14], making higher order transformation a valuable feature in any MDE-tool;

- *model differencing*: in order to support automated evolution on a industrial level, it must be possible to generate delta models out of two versions of a model. Moreover, it is desirable that the activity of meta-modelling does not have to change in order to support automated evolution. The Fujaba Difference Tool Suite uses the SiDiff framework to calculate and visualize the difference between two models' XMI documents [19]. A so-called Difference Viewer Plugin shows a coloured difference model in Fujaba;

- *transformation inversing*: in order to automatically co-evolve a transformation in domain evolution, the inverse of the migration transformation is needed. In Fujaba this is implicitly featured by providing the possibility to implement bidirectional transformations using Triple Graph Grammars (TGGs) in MoTE [6]. However, in that case, one is restricted to the use of bidirectional transformation with triple graph grammars. It remains an open question whether TGGs are expressive enough to obtain the inverse of the migration transformation (which may for example delete elements).

- *representation of semantics*: as not only the syntax but also semantics of a modelling language evolves, there must be a way to represent these semantic changes. A more precise means to reason about semantics preservation (through properties?) is needed.

If all of these pre-requisites are implemented in Fujaba, a framework for evolution can be relatively easily implemented.

6. CONCLUSIONS

Extensive adoption of model-driven engineering is obstructed by the lack of support for automated evolution. Especially in domain-specific modelling, modelling languages are used while under development or under ceaseless change. When such languages evolve, support for (semi-)automated co-evolution must be available. To this day, research has been done only to support model co-evolution for meta-model evolution. Transformations or semantics are not yet taken into account.

We addressed this problem by de-constructing all possible (co-)evolution processes into four basic scenarios, which can be combined. We showed that the co-evolution of transformations can be problematic, because a transformation always needs to be able to transform all possible elements in its domain.

We discussed the pre-requisites for an implementation of automated evolution. It turns out that, in order to implement support for evolution, a number of pre-requisites, such as higher order transformation, model differencing, transformation inversing and semantics represention, have to be dealt with. Like all other current tools, Fujaba only supports a few of these, with higher order transformation as major absentee.

7. ACKNOWLEDGMENTS

We would like to thank Tihamér Levendovszky for the fruitful discussions on the subject of this paper.

8. REFERENCES

[1] M. Alanen and I. Porres. Difference and union of models, 2003.

[2] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Rec.*, 16(3):311–322, 1987.

[3] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *EDOC '08: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society.

[4] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[5] H. Giese, T. Levendovszky, and H. Vangheluwe. Summary of the workshop on multi-paradigm modeling: Concepts and tools. In T. Kühne, editor, *Models in Software Engineering Workshops and Symposia at MoDELS 2006*, volume 4364 of *LNCS*, pages 252–262. Springer-Verlag, October 2006.

[6] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy*, volume 4199 of *Lecture Notes in Computer Science (LNCS)*, pages 543–557. Springer Verlag, 10 2006.

[7] B. Gruschko, D. Kolovos, and R. Paige. Towards synchronizing models with evolving metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution at IEEE European Conference on Software Maintenance and Reengineering (ECSMR)*, 2007.

[8] M. Herrmannsdoerfer, S. Benz, and E. Juergens. Cope - automating coupled evolution of metamodels and models. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 52–76, 2009.

[9] J. Hoessler, J. Soden, Michael, and H. Eichler. Coevolution of models, metamodels and transformations. *Models and Human Reasoning*, pages 129–154, 2005.

[10] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, March 2008.

[11] R. Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, Nov. 2004.

[12] R. Lämmel and W. Lohmann. Format Evolution. In *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.

[13] Y. Lin, J. Gray, and F. Jouault. Dsmdiff: A differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4, Special Issue on Model-Driven Systems Development):349–361, 2007.

[14] B. Meyers and P. Van Gorp. Towards a hybrid transformation language: Implicit and explicit rule scheduling in story diagrams. Sixth International Fujaba Days, September 18–19 2008.

[15] P. J. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling: An introduction. In *SIMULATION80*, volume 9, pages 433–450, 2004.

[16] O. Muliawan. Extending a model transformation language using higher order transformations. *Reverse Engineering, Working Conference on*, 0:315–318, 2008.

[17] D. Ohst, M. Welle, and U. Kelter. Differences between versions of uml diagrams. *SIGSOFT Softw. Eng. Notes*, 28(5):227–236, 2003.

[18] M. Pizka and E. Jurgens. Automating language evolution. In *TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 305–315, Washington, DC, USA, 2007. IEEE Computer Society.

[19] M. Schmidt and T. Gloetzner. Constructing difference tools for models using the sidiff framework. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 947–948, New York, NY, USA, 2008. ACM.

[20] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15, April 2004.

[21] S. Vermolen and E. Visser. Heterogeneous coupled evolution of software languages. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 630–644, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] G. Wachsmuth. Metamodel adaptation and model co-adaptation. In E. Ernst, editor, *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer-Verlag, July 2007.

[23] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[24] J. Zhang and J. Gray. A generative approach to model interpreter evolution. In *OOPSLA Workshop on Domain-Specific Modeling*, pages 121–129, 11 2004. Vancouver, VC.