Evolution of Modelling Languages

Bart Meyers[a], Raphael Mannadiar[b], Hans Vangheluwe[a],

[a]*Modelling, Simulation and Design Lab (MSDL), University of Antwerp, Middelheimlaan 1, B-2020 Antwerp, Belgium*
[b]*Modelling, Simulation and Design Lab (MSDL), McGill University, 3480 University Street, Montreal, Quebec, Canada*

Abstract

In model-driven engineering, evolution is inevitable over the course of the complete life cycle of complex software-intensive systems and more importantly of entire product families. 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 a 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. We suggest that our structured approach will enable the design of (semi-)automatic model evolution solutions. Finally, the pre-requisites to allow an easy implementation of this solution in an MDE tool are discussed.[1]

1. Introduction

In software engineering, the evolution of software artifacts is ubiquitous. These artifacts can be programs, data, requirements, documentation, and even 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]. A formal underpinning for DSM is given by multi-paradigm modelling (MPM) [15].

The high dependence on their domains and the need for rapid deployment make DSLs highly susceptible to change and incremental growth. 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 dealt with evolution issues manually. However, this 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 can not scale to industrial use.

The remainder of this paper is organised as follows: Section 2 is a quick introduction of modelling languages. Section 3 discusses related work. Section 4 presents the possible kinds of evolution. Section 5 introduces a way to tackle evolution of modelling languages by de-constructing the problem. Section 6 discusses the pre-requisites of an MDE tool that are necessary for implementing the framework for evolution. In Section 7 the paper is concluded.

2. 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 [6].

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.

[1] An early version of this work was previously presented at the Fujaba Days '09 workshop in Eindhoven, The Netherlands

Email addresses: Bart.Meyers@ua.ac.be (Bart Meyers), rmanna@cs.mcgill.ca (Raphael Mannadiar), Hans.Vangheluwe@ua.ac.be (Hans Vangheluwe)

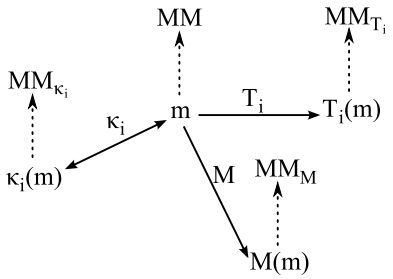

Figure 1: A model and its relations in MPM.

The abstract syntax contains the essence of the structure of the model (as an abstract syntax graph - as models are usually represented as graphs), which can be used as a basis for semantic anchoring [3].

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 at least be 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, "conforms to"-relationships are dotted arrows. The abstract syntax model m conforms to its meta-model MM. There is a bi-directional 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.

Because a modelled system consists of models written in their language and explicit relations between these models, the architecture of any modelled system can be mapped on (multiples of) Figure 1. Obviously, multiple models m can exist in a system, typically all conform to the same metamodel MM. Each model can be mapped on Figure 1 separately. A special case is multi-paradigm modelling, where models in different formalisms describe the system. The models are related however, so a transformation T (or semantic mapping M) is applied. As a consequence, $T(m)$ (or $M(m)$) can itself be mapped to an m in Figure 1, on which possibly relations like T and M apply. So, this is an horizontal extension of the system of Figure 1. Another special case is domain-specific modelling, where languages are explicitly modelled. So the metamodel MM of a DSL is in its own right a software artifact that's part of the system. In this sense, MM itself can be mapped to an m in Figure 1, having a metamodel of its own (e.g. entity-relationship diagrams). Also note that programs can be considered models, with an abstract syntax tree, a lexical concrete syntax and a semantic mapping to, for instance, machine code.

In conclusion, Figure 1 generalizes any modelled system, so a framework for evolution must support the possible scenarios that emerge from that diagram.

3. Related Work

In this section other work related to evolution is presented and some useful concepts are introduced. The related research fields that are discussed, namely model differencing and model co-evolution, have been substantially investigated already.

3.1. 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 actual 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, 5]. Existing approaches typically rely on the abstract syntax graphs (ASGs) of the two models to compare, 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 (or change-based) representation, the difference between two versions of a model is modelled as the series of edit operations (create/read/update/delete) that were performed 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 difficult to visualize. In structural (or state-based) representations, either the model (or its document object model (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 [5, 20].

3.2. 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, 4, 8], which we will call the *migration transformation*. Grushko et al. write this transformation manually using the Epsilon Transformation Language (ETL) [7].

Most 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.

Gruschko et al. make a distinction between non-breaking, resolvable and unresolvable operations. Non-breaking operations do not render models non-conformant to their meta-model, and hence do not require co-evolution. Inconsistencies caused by resolvable operations can be resolved by automated 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. To illustrate, the operations proposed by Cicchetti et al. [4] are shown in Table 1. Note the similarities with refactoring patterns.

4. 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.

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

Table 1: Evolution operations as presented in [4].

4.1. 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 3.2. 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 meta-model. As a consequence, these transformations also 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 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 *m* is trivial: related models (e.g., $T_i(m)$) can co-evolve by executing the appropriate transformations (e.g., T_i). Note, however, that a co-evolved model may itself be a meta-model thereby possibly triggering a number of further co-evolutions.

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 restricts a transformation evolution to remain compliant to its source and target meta-models, which may not be desired. 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. The aforementioned complications stem from two special cases of transformation evolution. Firstly, the evolution of the parsing or pretty printing mapping functions 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 bi-directional 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.

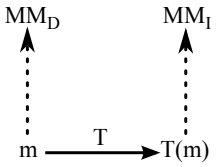

Figure 2: Basic co-evolution architecture.

4.2. Semantic Evolution

As mentioned above, semantics of a model are defined by its semantic mapping function to a semantic domain. Analyses can be performed on models in this semantic domain (e.g., deadlock detection in a Petri Net). The results of these analyses can be considered *properties* of the model, or $P_j(m)$. 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 is non null). 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. When two versions of a system are (a) equal modulo their intended syntactic and semantic changes and (b) syntactically consistent, the evolution of the system is said to be *continuous*. Only continuous evolutions are deemed correct (and meaningful).

5. De-constructing Evolution

As discussed in the previous section, there are infinitely many possible co-evolution scenarios. These possibilities can be mapped on the architectural model of a system of Figure 1. When we look more closely to the possible scenarios, similarities can be distinguished in the steps that are taken in the co-evolution scenarios. Because consistency and continuity must be preserved when the system evolves, co-evolution is in fact the (chained) interactive relationship between meta-model, model and transformation (instead of only meta-model and model, as suggested in related work). Thereby, it is important to note that unidirectional transformations have a domain, i.e. the formalism that is translated, and an image, i.e. the formalism this domain is translated to. In other words, for language evolution, both *incoming* and *outgoing* transformations must be dealt with. Therefore, we can break down the problem of co-evolution to the diagram in Figure 2, where a domain meta-model MM_D and an image meta-model MM_I, and there instance models are distinguished, with a transformation T in between. Again, arrows are transformations and dotted arrows are "conforms to"-relationships.

Now, these scenarios can always be broken down into a few basic ones which are depicted in Figure 3. Dashed arrows denote a (semi-)automatic generation. Each diagram starts from the relation between domain and image meta-models, given in Figure 2 (in bold font).

5.1. Case Study

We begin by introducing a simple example which we will come back to to illustrate the concepts below. Figure 4 shows a possible meta-model for a *Railroad* domain: `Rails`, `Junctions` and `Stations` can be connected into arbitrary networks on which `Trains` can circulate. Now, recall Figure 1: Figure 5 (a) shows the concrete syntax representation $\kappa_i(m)$ of a *Railroad* instance model; Figure 5 (b) shows the bi-directional transformation κ_i to (pretty printing) and from (parsing) the concrete syntax representation; Figure 5 (c) shows the abstract syntax representation m of the same *Railroad* instance model; Figure 5 (d) shows the semantic mapping M from the *Railroad* domain onto the Petri Net formalism; and Figure 5 (e) shows the semantic domain representation $M(m)$ of the same *Railroad* instance model. The example is simplified for clarity.

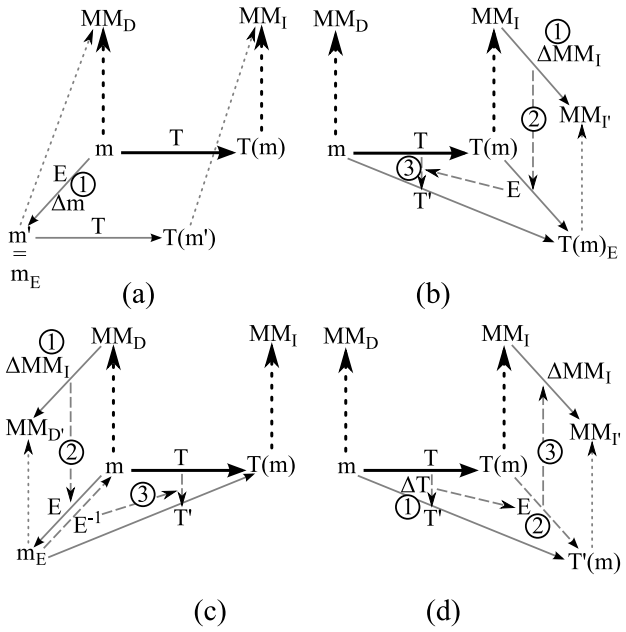

Figure 3: Co-evolution in (a) model evolution, (b) image evolution, (c) domain evolution and (d) transformation evolution.

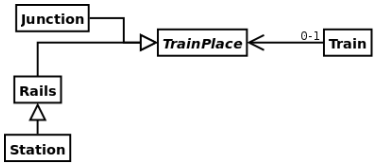

Figure 4: A meta-model for a *Railroad* domain.

5.2. Model Evolution

Figure 3 (a) shows model evolution. Some model m evolves to m'. In step 1 (the only step), a delta model Δm 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 previous equation can thus be rewritten as $m_E = m + \Delta m = m'$. As previously discussed, because m evolved to m', every transformation T must be executed again resulting in models $T(m')$ which conform to MM_i.

In the context of the *Railroad* example, models could evolve via the insertion, removal or redirection of `TrainPlaces` (which translates to changing the topography of the modelled railroad network) or `Trains` (which translates to changing the configuration of trains within the network). In both cases, it is trivial that the κ_i and M transformations (shown in Figure 5) as well as any other transformation T_i from model m would still be valid since no meta-model evolution has occurred. Hence, no further co-evolution is required.

5.3. Image Evolution

Image evolution is shown in Figure 3 (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 meta-models MM_I and $MM_{I'}$. In step 2, a migration transformation E is generated from ΔMM_I. The execution of E co-evolves models $T(m)$ to models $T(m)_E$ such that they conform to the new meta-model $MM_{I'}$. Moreover, the execution of transformation T has to result in valid models (i.e., models that conform to $MM_{I'}$). Consequently, T has to co-evolve to a new transformation T' (step 3), which is able to transform every possible model m conforming to MM_D to an appropriate model $T(m)_E$ conforming to $MM_{I'}$.

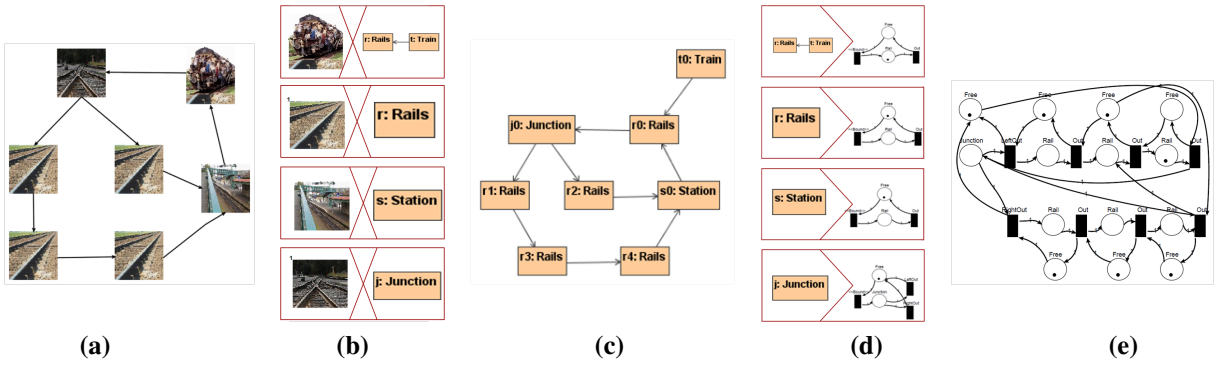

Figure 5: (a) The concrete syntax representation of a *Railroad* instance model; (b) The bi-directional abstract-concrete syntax transformation κ_i; (c) The abstract syntax representation of the same *Railroad* instance model; (d) The semantic mapping transformation *M* onto Petri Nets; (e) The semantic domain representation of the same *Railroad* instance model.

The diagram presents an intuitive solution for the generation of this *T'*: for every model *m*, *T'(m) = E(T(m))* holds, and thus we have $T' = E \circ T$. Hence, the co-evolved transformation *T'* can be obtained simply by composing transformations *T* and *E*.

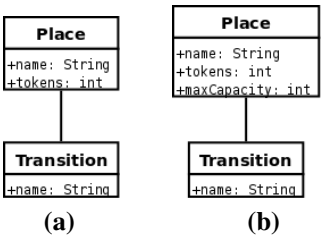

Figure 6: (a) The standard Petri Net formalism and (b) an evolved Petri Net formalism.

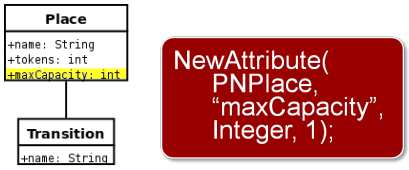

Figure 7: Two possible representations of the delta model describing the evolution of the Petri Net formalism.

In the context of the *Railroad* example, a possible image evolution would be the evolution of the target meta-model for the semantic mapping transformation (in this case, Petri Nets). Figure 6 shows the "standard" Petri Net formalism as well as a possible evolution where the notion of "capacity" (i.e., the maximum number of tokens allowed in a `Place` at any given time) is added to the `Place` construct. Figure 7 shows two possible representations, a coloured meta-model and a operational representation, of the associated delta model (step 1 from Figure 3 (b)). Figure 8 shows the migration transformation *E* that will translate standard Petri Nets to evolved ones (step 2 from Figure 3 (b)): in essence, the usual pattern for emulating the notion of capacity is matched and replaced by the newly added domain construct. Finally, the co-evolved transformation *T'* is in fact the new semantic mapping from *Railroad* instance models to semantically equivalent instance models of the evolved Petri Net formalism. As explained previously, *T'* can be expressed as $T' = E \circ T$ which, in this case, corresponds to the execution of the transformation from Figure 5 (d)

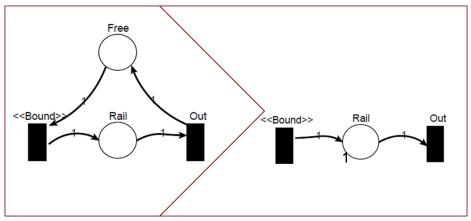

Figure 8: The migration transformation E that will translate "standard" Petri Nets to "evolved" ones.

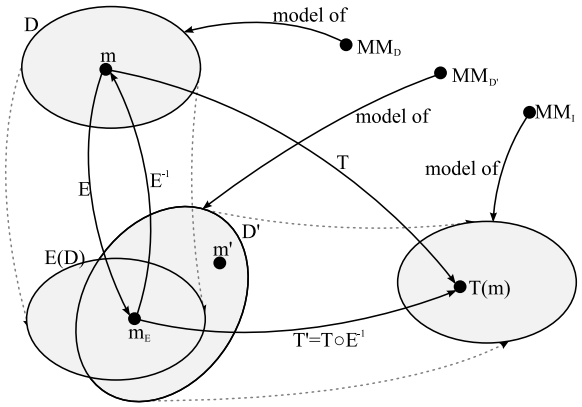

Figure 9: Set representation of domain-evolution. The evolution *E(D)* does not map onto *D'* exactly. For *m'*, the constraint $T' = T \circ E^{-1}$ does not hold!

followed by that of Figure 8. The resulting models will conform to the evolved Petri Net meta-model.

5.4. Domain Evolution

Figure 3 (c) shows domain evolution, where MM_D evolves. The artifacts that co-evolve are similar to those that do in the image evolution scenario. However, co-evolved transformations T' are now expressed as $T' = T \circ E^{-1}$ which implies the need to construct an inverse transformation E^{-1}. Unfortunately, the previous equation does not hold for the entire evolved domain D', as shown in Figure 9: the migration transformation E projects the entire domain D onto $E(D)$, but it is possible that $E(D) \neq D'$. In other words, though it may be possible to construct E^{-1} such that $T'(m_E) = T(E^{-1}(m_E))$ holds for a given model m in D, constructing such an inverse transformation for a model m' in D' may be impossible. Nevertheless, since transformation T' must apply to its entire domain D', the equation $T' = T \circ E^{-1}$ can not be used for all possible models conforming to $MM_{D'}$.

In the context of the *Railroad* example, a possible domain evolution would be the evolution of the *Railroad* domain itself. Figure 10 shows a possible evolution where the notion of "length" is added to the `Rail` construct. Figure 11 shows two possible representations of the associated delta model (step 1 from Figure 3 (c)). Figure 12 (a) shows the migration transformation E that will translate standard *Railroad* instance models to evolved ones (step 2 from Figure 3 (c)): in essence, sequences of connected `Rail`s are replaced by a single instance of the evolved `Rail` construct with the `length` field set to the length of the matched sequence. Figure 12 (b) shows E^{-1}, the inverse of migration transformation E, which translates evolved *Railroad* instance models back to standard ones by replacing the newly aggregated `Rail`s by appropriately sized sequences of the original `Rail` construct. Finally, the co-evolved transformation T', expressed as $T' = T \circ E^{-1}$, corresponds to the execution of the transformation from Figure 12 (b) followed by that of Figure 5 (d). The resulting models will conform to the standard Petri Net meta-model. Note that for the chosen evolution, the equation $E(D) = D'$ holds; i.e., the migration transformation can only produce syntactically correct and semantically equivalent instance models of the evolved *Railroad* domain and its inverse can only produce syntactically correct and semantically equivalent instance models of the original *Railroad* domain.

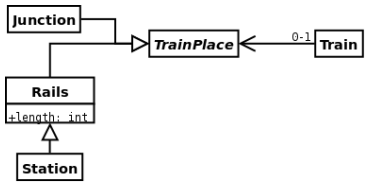

Figure 10: The evolved *Railroad* domain from Figure 4.

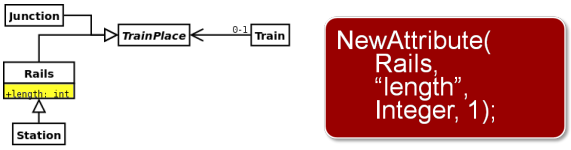

Figure 11: Two possible representations of the delta model describing the evolution of the *Railroad* formalism.

5.5. Transformation Evolution

Figure 3 (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 though, other artifacts such as the image meta-model might also co-evolve. In such cases, a migration transformation E must be constructed from which a delta model ΔMM_i can be derived. In the context of the *Railroad* example, a possible transformation evolution would be the evolution of the semantic mapping between the *Railroad* domain and the Petri Net formalism. It is clear that this would change the semantics of the *Railroad* domain.

5.6. Evolution Scenario Amalgamation

Using a combination of these four scenarios, all possible evolutions can be carried out. Note however that the problem of Figure 9 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 $dom_E(T) \nsubseteq dom(T)$.

In the context of the *Railroad* example, it is conceivable that a meta-modeller would require the proposed domain and image evolutions; i.e., that he would need to add a "length" attribute to the `Rail` construct and a "capacity" attribute to `Place` construct. As shown in Figure 13, merging our solutions for both of these basic evolution scenarios produces a sequence of three transformations that yield an evolved Petri Net instance model from a semantically equivalent evolved *Railroad* instance model. First, we execute the inverse migration transformation E^{-1} from Figure 12 (b) to produce a semantically equivalent model which conforms to the expected domain meta-model of the transformation T from Figure 5 (d). Second, we execute the transformation T from Figure 5 (d) to produce a semantically equivalent standard Petri Net. Finally, we execute the migration transformation E from Figure 8 to produce a semantically equivalent evolved Petri Net.

An obvious remark from the sequence of transformations depicted in Figure 13 is that simply combining solutions to basic scenarios to solve complex scenarios may yield sub-optimal results. In a system under development, an additional transformation must be executed for each additional evolution. Hence, the total number of migration transformations to be executed after an evolution is $O(m \cdot v)$, with v the number of existing versions and m the number of artifacts that are co-evolved. It may be possible to obtain the same output models while executing less and/or simpler transformation rules. An open research problem is then the (semi-)automatic merging of sequences of transformations into more efficient and ideally optimal sets of transformation rules.

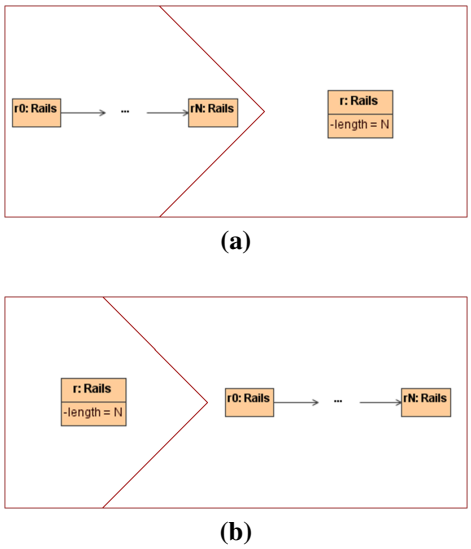

Figure 12: (a) The migration transformation E that will translate "original" *Railroad* instance models to "evolved" ones and (b) the inverse transformation E^{-1}.

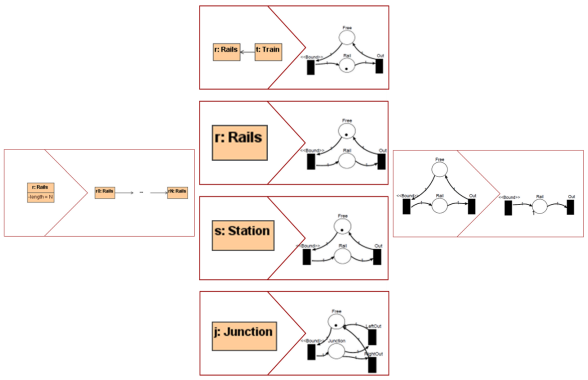

Figure 13: The amalgamation of the solutions for the example image and domain evolutions.

6. Pre-requisites for Evolution

Following the discussion above, the proposed approach depends on a few more general techniques. However, these techniques are not yet featured in the average MDE tool and as a consequence, implementing all forms of evolution depends on the implementation of these pre-requisites. 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. In order to support higher order transformations, the transformation language must be modelled explicitly (i.e., the meta-model is not available). Several other uses for higher order transformation, inside and out of the context of evolution, are discussed in [4, 16, 14]. It is widely accepted that higher order transformation are 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. Difference tools such as DSMDiff [13] and UMLDiff [23] are useful as they detect differences retrospectively (see also Section 3.1);

- *transformation inversing*: in order to automatically co-evolve a transformation in domain evolution, the inverse of the migration transformation is needed. This can be implicitly featured by providing the possibility to implement bi-directional transformations using Triple Graph Grammars (TGGs). However, in that case, one is restricted to the use of bi-directional 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 the 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 is needed, like for example using the properties of a model.

- *merging of transformations*: in order to optimize the sequences of transformations, they are merged. In this sense, automatic merging of transformations can be of great convenience.

If all of these pre-requisites are supported by your MDE tool of choice, a framework for evolution can be relatively easily implemented.

7. Conclusions

Extensive adoption of model-driven engineering is obstructed by the lack of support for automated evolution. In domain-specific modelling especially, modelling languages are used while under development or under on-going 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 and illustrated this problem by de-constructing all possible (co-)evolution scenarios into four basic cases, which can each be handled (semi-)automatically. 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.

Finally, 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, semantics representation and transformation merging have to be dealt with. If these techniques are available in an MDE tool, a framework for (semi-)atomatic evolution can be easily implemented.

8. Bibliography

[1] Alanen, M., Porres, I., 2003. Difference and union of models.
URL `http://citeseer.ist.psu.edu/596185.html`

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

[3] Chen, K., Sztipanovits, J., Neema, S., 2005. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In: EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software. ACM, New York, NY, USA, pp. 35–43.

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

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

[6] Giese, H., Levendovszky, T., Vangheluwe, H., October 2006. Summary of the workshop on multi-paradigm modeling: Concepts and tools. In: Kühne, T. (Ed.), Models in Software Engineering Workshops and Symposia at MoDELS 2006. Vol. 4364 of LNCS. Springer-Verlag, pp. 252–262.

[7] Gruschko, B., Kolovos, D., Paige, R., 2007. 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).

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

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

[10] Kelly, S., Tolvanen, J.-P., March 2008. Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons.
URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0470036664

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

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

[13] Lin, Y., Gray, J., Jouault, F., 2007. Dsmdiff: A differentiation tool for domain-specific models. European Journal of Information Systems 16 (4, Special Issue on Model-Driven Systems Development), 349–361.
URL http://www.cis.uab.edu/gray/Pubs/ejis-2007.pdf

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

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

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

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

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

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

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

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

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

[23] Xing, Z., Stroulia, E., 2005. Umldiff: an algorithm for object-oriented design differencing. In: ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, New York, NY, USA, pp. 54–65.
URL http://dx.doi.org/10.1145/1101908.1101919

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