Exploring DSL Evolutionary Patterns in Practice
A study of DSL evolution in a large-scale industrial DSL repository

Josh G.M. Mengerink[1], Bram van der Sanden[1], Bram C.M. Cappers[1], Alexander Serebrenik[1], Ramon R.H. Schiffelers[1,2] and Mark G.J. van den Brand[1],

[1]*Eindhoven University of Technology, The Netherlands*

[2]*ASML, Veldhoven, The Netherlands*

{*j.g.m.mengerink, b.v.d.sanden, b.c.m.cappers, a.serebrenik, r.r.h.schifflers, m.g.j.v.d.brand*}*@tue.nl,*
ramon.schiffelers@asml.com

Keywords: Model Driven Engineering, Evolution, Maintenance

Abstract: Model-driven engineering is used in the design of systems to (a.o.) enable analysis early in the design process. For instance, by using domain-specific languages, enabling engineers to model systems in terms of their domain, rather then encoding them into general purpose modeling languages. Domain-specific languages, like classical software, evolve over time. When domain languages evolve, they may trigger co-evolution of models, model-to-model transformations, editors (both graphical and textual), and other artifacts that depend on the domain-specific language. This co-evolution can be tedious and very costly.

In literature, various approaches are proposed towards automated co-evolution. However, these approaches do not reach full automation. Several other studies have shown that there are theoretical limitations to the level of automation that can be achieved in certain scenarios. For several scenarios full automation can never be achieved. We wish to gain insight to which extent practically occurring scenarios can be automated.

To gain this insight, in this paper, we investigate on a large-scale industrial repository, which (co-)evolutionary scenarios occur in practice, and compare them with the various scenarios and their theoretical automatability. We then assess whether practically occurring scenarios can be fully automated.

1 An Introduction to Model Co-Evolution

Model-driven software engineering (MDSE) has many promises, such as improved productivity and quality. Among those strengths is the ability to perform analysis early in the design process (Schiffelers et al., 2012), (Karsai et al., 2003). For instance, computing the expected throughput of a modeled logistic system, giving feedback earlier in the design process and increasing productivity.

One way to support such early analysis is to design domain-specific languages (DSLs) enabling engineers to model (parts of) systems in terms close to their knowledge domain. Subsequently, those models can be transformed into dedicated analysis formalisms (Lara and Vangheluwe, 2002) such as mCRL2 (Groote et al., 2007), UPPAAL (Bengtsson et al., 1995), or SDF (SDF, 2015) using standard model-to-model transformation techniques (*e.g.,* QVT or ATL (QVT, 2015; Jouault and Kurtev, 2006; Kolovos et al., 2008)). Such a two-phase approach (transformation from specification models to analyses models) has been successfully implemented in various studies in industry (Schiffelers et al., 2012), (Mohagheghi et al., 2013).

However, MDSE using DSLs also has challenges. As metamodels (which underpin DSLs) have become the central artifact in the design (Mengerink et al., 2016), (Vissers et al., 2016), their evolution (Favre, 2005) can trigger forced co-evolution of other ecosystem artifacts such as editors (Di Ruscio et al., 2011), constraints (Khelladi et al., 2016), transformations (García et al., 2013; Levendovszky et al., 2010), and models (Gruschko et al., 2007; Narayanan et al., 2009; Cicchetti et al., 2008; Wachsmuth, 2007). This process is similar to changing source code in response to API evolution in a traditional software engineering context (Dig and Johnson, 2005), and results in a tedious, error-prone, and thus costly process of co-evolution.

Various approaches have been proposed aiming to (partially) automate co-evolution of metamodel-dependent artifacts (Khelladi et al., 2016; García

et al., 2013; Levendovszky et al., 2010; Di Ruscio et al., 2011). In particular, co-evolution of models has received significant attention (Rose et al., 2009; Wachsmuth, 2007; Gruschko et al., 2007; Di Rocco et al., 2012), as models are most numerous artifacts depending on the metamodel (Vissers et al., 2016). Furthermore, several studies have also approached the co-evolution problem from a theoretical stance (Hermannsdörfer and Ratiu, 2009; Sprinkle et al., 2009), determining the limits of automation. These papers argue that when the constituting parts (*e.g.,* syntax and semantics) of DSLs evolve, full automation can only be achieved in certain cases [1].

In our research, we are working towards improved automation of model co-evolution, but wonder to what extent this is feasible. To ascertain the impact of the fundamental limitations (Sprinkle et al., 2009) in practice, we wish to understand which of these cases occur in practice.

The remainder of this paper is structured as follows: we describe our case study and experiments in Section 2 and discuss the results in Section 3. We discuss threats to validity and conclusions in Sections 4 and 5 respectively.

2 Study

2.1 Evolutionary Patterns

To understand what evolutionary patterns exist in terms of a DSL, we first identify the various components of a DSL. Sprinkle *et al.* (Sprinkle et al., 2009) have provided a formal definition of a DSL as a tuple of:

1. Abstract Syntax (*i.e.,* the metamodel), denoted $\mathcal{A}$;
2. Constraints (*e.g.,* OCL constraints on the metamodel), denoted $\mathcal{C}$;
3. A semantic domain, denoted $\mathcal{SD}$;
4. Semantics that map syntax to the semantic domain, denoted $\mathcal{S}$.

We then define an evolutionary pattern as a change of any combination of (1), (2), (3), and/or (4). For example: syntax only, or syntax and semantics co-evolution.

[1] Throughout this paper, the terms cases and patterns both refer to a particular co-evolution of constituent parts of a DSL

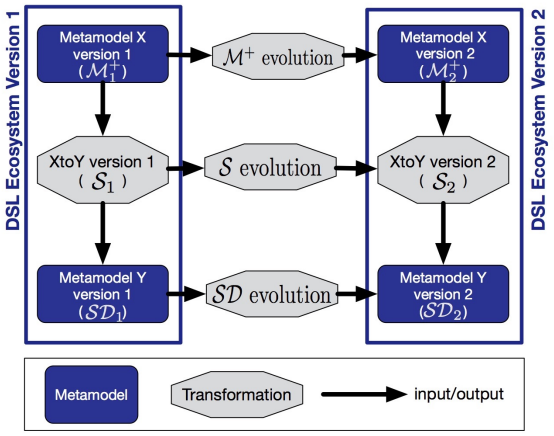

Figure 1: The relationship between the formal definition of a language, and various artifacts in a repository.

2.2 Industrial Context

Our research takes place at ASML, provider of lithography equipment for the semiconductor industry. At ASML, MDE is used to enable virtual prototyping to provide feedback early in the design process. In particular, we perform our case study (*cf.* (Runeson and Höst, 2008)) on the CARM ecosystem of DSLs (Schiffelers et al., 2012). The CARM ecosystem consists of more than 20 DSLS with over 100 model-to-model transformations, with a revision history of up to six years. Note that we only study files committed to the main branch of this repository, as subsequent states of this branch represent finished products.

In CARM (Schiffelers et al., 2012), we observe that the definition of a DSL as described by Sprinkle *et al.* (Sprinkle et al., 2009) is not present in practice. Rather than the decomposition described in Section 2.1, we see that the abstract syntax ($\mathcal{A}$) and constraints ($\mathcal{C}$) are often combined into a single metamodel specification (denoted $\mathcal{M}^+$).

Furthermore, the semantic domain ($\mathcal{SD}$) is implemented as a second metamodel. Often, this is the metamodel of an analysis DSL, rather than some abstract domain with a notion of equivalence (as the description of Sprinkle *et al.* (Sprinkle et al., 2009) suggests). Lastly, the model-to-model transformation between the specification DSL metamodel (*i.e.,* $\mathcal{M}^+$) and analysis DSL metamodel ($\mathcal{SD}$) takes on the role of $\mathcal{S}$. These relations (and their evolution) are illustrated in Figure 1.

Observing this practical organisation of DSLs at ASML, we limit ourselves to combinations of evolutions of:

1. Syntax-defining metamodel, $\mathcal{M}^+$;
2. The semantics-defining transformation, $\mathcal{S}$;

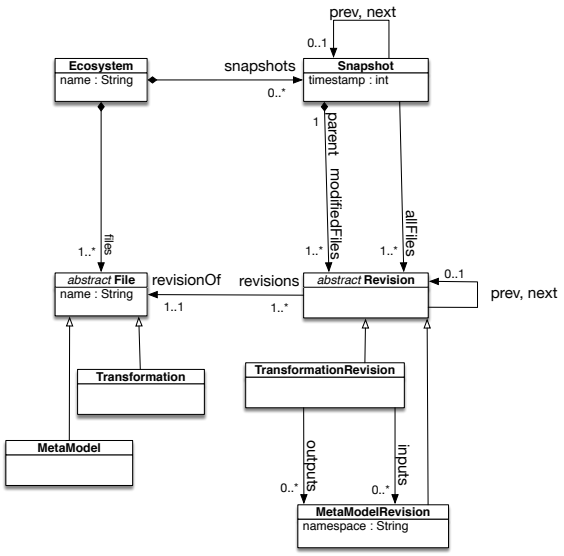

Figure 2: A graphical representation of the metamodel we use to specify the evolution history of our ecosystem.

3. The metamodel defining the semantic domain, $\mathcal{SD}$.

This results in a total of seven cases ($= 2^3 - 1$), which excludes the pattern where nothing evolves.

2.3 Experimental Setup

In the spirit of MDE, we reconstruct the evolution history of DSLs as a model. For this purpose we have designed the metamodel illustrated in Figure 2. On this metamodel, we subsequently define patterns of interest.

The all-encompassing concept in the metamodel is the `Ecosystem`. An ecosystem consists of various `File`s such as `MetaModel`s and `Transformation`s. As the files evolve over time, `Snapshot`s represent the state of the ecosystem at a particular point in time, as indicated in the `timestamp` attribute. In a snapshot, various files are modified, as encoded in the `modifiedFiles` reference. A snapshot also has a reference to files that were "carried over" from earlier revisions, by means of the `allFiles` reference.

`Revision` is a version of a file at a particular point in time. Similar to the distinction between two types of `File`: `Metamodel`s and `transformation`s, we distinguish between `MetaModelRevision`s and `TransformationRevision`s. Since the `namespace` of a DSL, as well as the input and the output of the transformations, can evolve in time, we represent them as attributes of `MetaModelRevision`s and `TransformationRevision`s rather than `Metamodel` and `Transformation`.

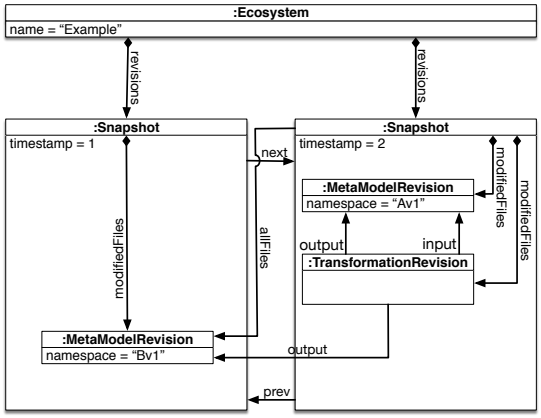

Figure 3: An object diagram (also known as instance diagram) of a model conforming to the metamodel in Figure 2. Specifically, an instance of Ecosystem is presented. For clarity, several edges have been omitted but do recall that *Snapshot.modifiedFiles* $\subseteq$ *Snapshot.allFiles*.
This model shows two subsequent snapshots of the repository (at times $t = 1$ and $t = 2$). In snapshot 1, DSL B is modified, in snapshot 2, DSL A and a transformation from A to B are modified.

Subsequently, we can reconstruct the revision history as a model in our metamodel. In Figure 3, we present an example fragment of such a model. As stated in Section 2.2, the constituting parts of our DSL are in practice present as two DSLs A and B (representing $\mathcal{M}^+$ and $\mathcal{SD}$) and a transformation between them (representing $\mathcal{S}$). Such a pattern is also illustrated in Figure 3, where we see the pattern of an evolving syntax and semantics, but the semantic domain does not evolve. From the reconstructed model of our repository, we wish to extract for evolutionary pattern, how many `Revision`s evolve according to that pattern.

To encode the various combinations in which syntax, semantics, and semantic domain can change, we create `configurations`, which are boolean triples encoding the particular combination of interest. For every combination we create one such triple:

$$\texttt{Configuration} : \mathbb{B} \times \mathbb{B} \times \mathbb{B}$$

For example, the pattern in which syntax changes, semantic changes, but the semantic domain does not, is encoded as $\langle true, true, false \rangle$. We also refer to this configuration as Q110. The only configuration which we are not interested in is $\langle false, false, false \rangle$, as there is no (co-)evolution.

Note that the common ground between every configuration of interest is that each one has at least one artifact that evolves. In terms of our model this means that we can iterate over every `Snapshot`, and at least one artifact relevant for the pattern is in the

```
Data: An ecosystem e, and a configuration
      Conf
Result: A collection of all revisions that play a
        part in Conf
results ← ∅
foreach snapshot : Snapshot ∈ e.snapshots do
    foreach revision : Revision ∈
     snapshot.modifiedFiles do
        if evolvesAsPattern(revision, Conf)
         then
            results ← results ∪ ⟨Conf, revision⟩
        end
    end
end
return results
```

Algorithm 1: Iterate over every `Revision` to see if it plays a part in a particular pattern. We can iterate over `modifiedFiles` only, because every `Revision` is uniquely contained in the `modifiedFiles` relation of a `Snapshot`.

`modifiedFiles` of that snapshot. This knowledge is used in Algorithm 1.

This leaves the problem of identifying whether a revision plays a part in a pattern as described by a configuration. This identification is given by means of the *evolvesAsPattern* function:

$$\text{evolvesAsPattern}(r : \mathit{Revision}, c : \mathit{Configuration}) = \text{evolvesAsSyntax}(r,c) \vee \text{evolvesAsSemantics}(r,c) \vee \text{evolvesAsSemanticDomain}(r,c)$$

It is important to note that we consider all input metamodels of a transformation together to specify syntax, and all output metamodels of a transformation together to specify the semantic domain. This means that:

- The syntax, or semantic domain is considered to change, if any of its composing `MetaModelRevision`s change.
- The syntax, or semantic domain is considered to stay the same if none of its composing `MetaModelRevision`s change.

For more informations about the constituent functions, we refer to the algorithms in the appendix. Algorithm 2 for *evolvesAsSyntax*, 3 for *evolvesAsSemantics*, and 4 for *evolvesAsSemanticDomain*.

The result from Algorithm 1 then contains, per configuration, all the revisions of files that evolve according to that configuration. The results of this analysis are presented in Section 3.

2.4 Mining Git

In order to perform our analyses, we extract relevant files the git version-control system at ASML. At ASML, the GIT repository makes use of a master branch that represents finished states. Features are developed in separate branched and merged into master once completed. In this work, we limit ourselves to the main branch of the repository as commits to this branch should represent the full intended change to the ecosystem.

On the main branch, starting with the earliest commit, we look at subsequent commits to the ecosystem and determine which files were added, changed, or deleted. This earliest-first approach allows us to deal with merges as regular commits.

2.5 Instance Model Creation

To create an instance of the metamodel in Figure 2, we take the following steps. Starting with the earliest commit to the main branch, we create a snapshot S_0 for this commit. All files in that commit are added to both the "modifiedFiles" and 'allFiles' of S_0.

For every subsequent commit (including merges) to the main branch ,we create a snapshot S_i with a timestamp corresponding to the commit. Subsequently:

- Every file added or modified in that commit is added to both the "modifiedFiles" and 'allFiles' of S_i.
- Files from the "allFiles" of the previous version (S_{i-1}) that were not added/modified/deleted in this commit are added to the "allFiles" of S_i.

For subsequent snapshots, the "next" and "prev" relations are set.

Once the above relations have all been initialized, we can begin creating the "input" and "output" relations of the various transformations as described in Section 2. For instance, in the example illustated in Figure 3, we parse a TransformationRevision, and find that it has "Av1" and "Bv1" as input and output respectively. As the TransformationRevision resides at timestamp $t = 2$, we have to search backwards (starting at $t = 2$) through all snapshots for MetaModelRevisions with the appropriate names. In Figure 3, this means that we use the "Av1" at time $t = 2$, and "Bv1" at time $t = 1$.

After the "input" and "output" relations have been properly created, we can begin our pattern-analysis.

Table 1: Number of revisions evolving according to a particular configuration

Reference	$\mathcal{M}^+$ evolves	$\mathcal{S}$ evolves	$\mathcal{SD}$ evolves	#
Q010	no	yes	no	865
Q100	yes	no	no	368
Q001	no	no	yes	344
Q111	yes	yes	yes	296
Q110	yes	yes	no	86
Q011	no	yes	yes	84
Q101	yes	no	yes	0

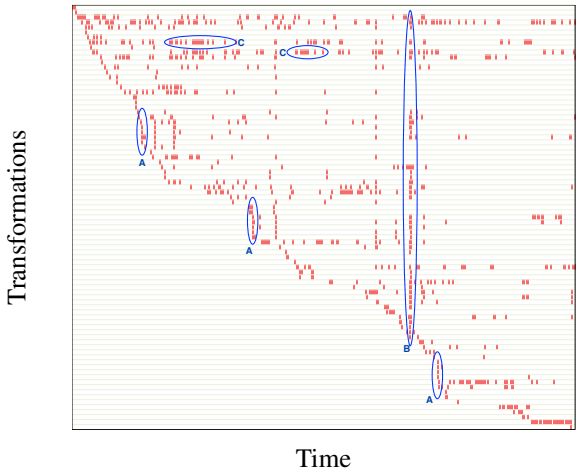

Figure 4: Illustration showing the modification of model-to-model transformations over time (Q010). Each row represents a transformation, and each column represents a moment in time. Columns are sorted chronologically, and rows are sorted by initial creation of the transformation. A cell is colored red if the given transformation changed at that moment in time. Names of transformations have been omitted for reasons of confidentiality.
Observe that various transformations are often created together (A), and maintained together (B). Also, once a transformation is modified, various revisions ensue in rapid succession (C).

3 Discussion of Results

Using the analysis from Section 2, we have obtain the results which are presented in Table 1. In Sections 3.1 through 3.4 we discuss the most frequently occuring cases in more detail.

3.1 Redefinition of Semantics (Q010)

As the semantics-only DSL evolution case (Q010) has the most occurences, we can say that the semantic definitions of DSLs are the most volatile part. Plotting the changes over time as illustrated in Figure 4, we can see that the evolution frequency of the various transformations seems to decrease over time. This enforces our intuition that DSL semantics stabilize over time.

Interesting in Figure 4 is the large vertical column (encircled in blue), in which a majority of transformation seems to have been updated. Upon closer (manual) inspection, a new website for documentation was introduced, and all error messages were refactored to include appropriate links to this site. This, in essence, was not an update to the semantics. These log-related updates appear to be the only non-semantical updates performed.

To ensure that these non-semantical updates do now invalidate our findings, we further analyzed to what extent they are present in our data. An analysis of these non-semantical updates shows that of the 17187 differences that were observed 57 are creations of new log-statements, 201 are modifications of existing log-expressions, and 1 removal of a log statement. The total of 259 non-semantical changes thus constitutes approximately 1.5% (259/17187) of the observed data, which leads us to conclude that our results are not invalidated by presence of non-semantical changes.

The observed volatility (the fact that most DSL evolutions are semantics-based), creates the necessity to incorporate semantics into the co-evolution process of models. Interestingly, the work of Sprinkle *et al.* (Sprinkle et al., 2009) does not discuss the fundamental limitations of model co-evolution in response to evolution of only DSL semantics.

The question we remain with is: given a model m_1 for $\mathcal{M}^+$, does a model m_2 exists that has equivalent semantics using the new semantical definition $\mathcal{S}_2$?

$$\exists_{m_2 \in \mathcal{M}_2} [\mathcal{S}_2(m_2) \equiv \mathcal{S}_1(m_1)]$$

In practice, we observe that identity $=$ is frequently used as a notion of equivalence ($\equiv$). In this case, existence of such an m_2 basically boils down to the question if the image ($\mathbb{I}$) of the original semantic function is a subset of the image of the updated semantic function:

$$\mathbb{I}(\mathcal{S}_1) \subseteq \mathbb{I}(\mathcal{S}_2)$$

In general, this problem is undecidable. Take for instance $\mathbb{I}(\mathcal{S}_1)$ to be $\langle true, false \rangle$, and let $\mathcal{S}_2$ map programs to true, if and only if that program terminates (the Halting problem). As the Halting problem is undecidable, as a result $\mathbb{I}(\mathcal{S}_2)$ is undecidable. Hence, it is undecidable whether $\mathbb{I}(\mathcal{S}_1) \subseteq \mathbb{I}(\mathcal{S}_2)$.

An interesting piece of future work would be to further investigate the nature of the $\mathcal{S}_1$ to $\mathcal{S}_2$ evolution.

3.2 Syntax-only DSL evolution (Q100)

The second most frequent occurring change are syntactic only in nature (Q100). In total, 368 revisions

evolved according to this pattern. We have already studies the different kinds of syntactic changes that typically occur in practice (Mengerink et al., 2016), (Vissers et al., 2016). Furthermore, various approaches exist that perform syntactic model co-evolution in response to syntax-only DSL evolution (Eda, 2015; Rose et al., 2010; Di Rocco et al., 2012).

Upon more detailed study of the automation of this case, Sprinkle *et al.* (Sprinkle et al., 2009) argue that semantic model co-evolution is possible in a large number of cases. More specifically, for additive evolutions, it is always possible, and for subtractive changes it depends on the specific context.

In our previous work (Mengerink et al., 2016) we have observed that additive change make up approximately 37% of syntactic DSL evolution, and that 22% of syntax evolutions is subtractive in nature. Moreover, we have looked into *changes* (*i.e.,* a change to a value in the metamodel is neither additive nor subtractive). This category makes up the remaining 41% of DSL evolutions. Additional research is required to ascertain the automatability of these syntax changes.

3.3 Evolution of the semantic domain (Q001)

The third most frequent case is evolution of the semantic domain. We expected the semantic domains (*i.e.,* our analysis DSLs) to be fairly stable over time. The number of occurences (344) were nearly as numerous as the syntactic evolutions (368), which surprised us. This led us to investigate the evolution of semantic domain DSLs further.

When intersecting all revisions that evolve as syntax (Q100) and the revisions that evolve as a semantic domain (Q001), we observe 309 revisions are in both sets. Upon further inspection, it turns out that transformations are often multi-step. That is, a DSL *A* is transformed into a DSL *B*, which in turn is transformed into a DSL *C*. In this case, when *B* evolves, it acts both as semantic domain, and as a syntax definition. Please note that this is distinct from pattern Q101.

3.4 Everything evolves (Q111)

The last major case (with over 200 occurrences), is Q111. Here, we have to acknowledge a threat to validity, as distinct evolutions may be obscured, *e.g.,* by our interpretation of git merges, skewing the observations we make. The number of occurrences in Q111 may thus be attributed to various other cases, meaning that the various other cases become more significant. However, to confirm this, more in-depth research is needed, which we mark as future work.

4 Threats to Validity

As with any repository mining work, there are some inherent threats (Bird et al., 2009). In particular, our assumption that a commit corresponds to a piece of work may lead to skewing of our results. However, as the set of files were consistent after every commit, we have confidence that this is not the case. Nonetheless, we envision future work to further analyze whether our results are indeed skewed.

Furthermore, we have inspected the change of a file, but not how that file was changed. This leads to some additional false positives, as described in Section 3.1. Further research is needed to perform a more in-depth analysis of the various changes.

5 Conclusion

In this work, we have investigated the evolution of DSLs in a large-scale industrial MDSE ecosystem. We conclude that, in our case study, of the various constituent parts of a DSL that can evolve:

> The most common type of DSL evolution is redefinition of its semantics.

Also, by mapping the automatability of semantic-preserving co-evolution to the Halting problem, we have argued that

> Automatability of semantic-preserving model co-evolution in response to DSL semantic evolution is undecidable in general.

Further analysis of the semantic redefinition of languages over time (Figure 4 showed that, in our case-study, the frequency of changes per transformation seems to decrease over time (barring exceptions). Leading us to conjecture:

> DSL semantics stabilize over time.

Lastly, we feel that this work shows that the evolution of DSL semantics plays a more dominant role in DSL evolution, where in literature, syntax is often assumed to be the more dominant.

More research into the role of DSL semantics in
(co-)evolution is needed

As future work, we envision investigating whether certain types always precede/succeed each other. For instance, is syntax evolution normally followed by semantic redefinition, rather than changing the syntax and semantics together in one go. Additionally, as stated in Section 3.1, there is a need to investigate the nature of DSL semantic change, in order to ascertain if for some of these changes automation of model of co-evolution is possible. Furthermore, additional case studies are necessary to determine if the results in this study can be generalized.

REFERENCES

(2015). Edapt. https://www.eclipse.org/edapt/. Accessed: 2015-04-07.

(2015). QVTo. http://www.eclipse.org/mmt/?project=qvto. Accessed: 2015-04-07.

(2015). SDF. http://www.es.ele.tue.nl/sdf3/. Accessed: 2015-04-07.

Bengtsson, J., Larsen, K. G., Larsson, F., Pettersson, P., and Yi, W. (1995). UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop*, pages 232–243.

Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., Germán, D. M., and Devanbu, P. T. (2009). The promises and perils of mining git. In *MSR*, pages 1–10.

Cicchetti, A., Di Ruscio, D., Eramo, R., and Pierantonio, A. (2008). Automating co-evolution in model-driven engineering. In *IEEE Enterprise Distributed Object Computing Conference*, pages 222–231.

Di Rocco, J., Iovino, L., and Pierantonio, A. (2012). Bridging state-based differencing and co-evolution. In *Models and Evolution*, pages 15–20. ACM.

Di Ruscio, D., Lämmel, R., and Pierantonio, A. (2011). Automated co-evolution of GMF editor models. In *SLE*, volume 6563 of *LNCS*, pages 143–162. Springer.

Dig, D. and Johnson, R. (2005). The role of refactorings in api evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398.

Favre, J.-M. (2005). Languages evolve too! changing the software time scale. In *Principles of Software Evolution*, pages 33–42.

García, J., Diaz, O., and Azanza, M. (2013). Model transformation co-evolution: A semi-automatic approach. In *SLE*, volume 7745 of *LNCS*, pages 144–163. Springer.

Groote, J. F., Mathijssen, A., Reniers, M., Usenko, Y., and van Weerdenburg, M. (2007). The formal specification language mcrl2. In *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings. Dagstuhl.

Gruschko, B., Kolovos, D., and Paige, R. (2007). Towards synchronizing models with evolving metamodels. In *Workshop on Model-Driven Software Evolution*.

Herrmannsdörfer, M. and Ratiu, D. (2009). Limitations of automating model migration in response to metamodel adaptation. In *MSE, Workshops and Symposia at MODELS*, volume 6002 of *LNCS*, pages 205–219. Springer.

Jouault, F. and Kurtev, I. (2006). Transforming models with atl. In Bruel, J.-M., editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 128–138. Springer.

Karsai, G., Sztipanovits, J., Ledeczi, A., and Bapty, T. (2003). Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164.

Khelladi, D. E., Hebig, R., Bendraou, R., Robin, J., and Gervais, M.-P. (2016). Metamodel and constraints co-evolution: A semi automatic maintenance of OCL constraints. In *ICSR*, pages 333–349. Springer.

Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2008). *The Epsilon Transformation Language*, pages 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg.

Lara, J. d. and Vangheluwe, H. (2002). Atom3: A tool for multi-formalism and meta-modelling. In Kutsche, R.-D. and Weber, H., editors, *FASE*, pages 174–188, Berlin, Heidelberg. Springer Berlin Heidelberg.

Levendovszky, T., Balasubramanian, D., Narayanan, A., and Karsai, G. (2010). A novel approach to semi-automated evolution of dsml model transformation. In *SLE*, volume 5969 of *LNCS*, pages 23–41. Springer.

Mengerink, J. G. M., Serebrenik, A., Schiffelers, R. R. H., and van den Brand, M. G. J. (2016). A complete operator library for DSL evolution specification. In *ICSME 2016, Raleigh, NC, USA*, pages 144–154.

Mohagheghi, P., Gilani, W., Stefanescu, A., Fernandez, M. A., Nordmoen, B., and Fritzsche, M. (2013). Where does model-driven engineering help? experiences from three industrial cases. *Software & Systems Modeling*, 12(3):619–639.

Narayanan, A., Levendovszky, T., Balasubramanian, D., and Karsai, G. (2009). Automatic domain model migration to manage metamodel evolution. In *MoDELS*, volume 5795 of *LNCS*, pages 706–711. Springer.

Rose, L. M., Kolovos, D. S., Paige, R. F., and Polack, F. A. (2010). Model migration with Epsilon Flock. In *ICMT*, volume 6142 of *LNCS*, pages 184–198. Springer.

Rose, L. M., Paige, R. F., Kolovos, D. S., and Polack, F. A. C. (2009). An analysis of approaches to model migration. In *MoDSE-MCCM*, pages 6–15.

Runeson, P. and Höst, M. (2008). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131.

Schiffelers, R. R. H., Alberts, W., and Voeten, J. P. M. (2012). Model-based specification, analysis and synthesis of servo controllers for lithoscanners. In *6th International Workshop on Multi-Paradigm Modeling*, pages 55–60. ACM.

Sprinkle, J., Gray, J., and Mernik, M. (2009). Fundamental limitations in domain-specific modeling language evolution. Technical report, Technical Report# Tr-0908311, University of Arizona.

Vissers, Y., Mengerink, J. G. M., Schiffelers, R. R. H., Serebrenik, A., and Reniers, M. A. (2016). Maintenance of specification models in industry using edapt. In *FDL*, pages 1–6.

Wachsmuth, G. (2007). Metamodel adaptation and model co-adaptation. In *ECOOP*, volume 4609 of *LNCS*, pages 600–624. Springer.

APPENDIX

Data: A revision r, and a configuration $Conf = \langle C_x, C_y, C_z \rangle$
Result: true iff r fulfills the role of syntactic definition and adheres to $Conf$

if $\neg(r$ *instanceof* `MetaModelRevision`$)$ **then**
  return *false*
end
$S_{curr} \leftarrow r.parent$
if C_y **then**
  $Q \leftarrow S_{curr}.modifiedFiles$
else
  $Q \leftarrow S_{curr}.prev.allFiles$
end
if C_z **then**
  $result \leftarrow result \wedge \exists_{y:\text{TransformationRevision}} [x \in input(y) \wedge y \in Q \wedge output(y) \cap S_{curr}.modifiedfiles \neq \emptyset]$
else
  $result \leftarrow result \wedge \exists_{y:\text{TransformationRevision}} [x \in input(y) \wedge y \in Q \wedge output(y) \neq \emptyset \wedge output(y) \cap S_{curr}.modifiedfiles = \emptyset]$
end
return *result*

Algorithm 2: Checks if a given `Revision` is a syntactic definition that adheres to a provided configuration.

Data: A revision r, and a configuration $Conf = \langle C_x, C_y, C_z \rangle$
Result: true iff r fulfills the role of semantic mapping

if $\neg(r$ *instanceof* `TransformRevision`$)$ **then**
  return *false*
`/* Avoid empty domain rule */`
$result \leftarrow (inputs(y) \neq \emptyset) \wedge (outputs(y) \neq \emptyset)$
$S_{mod} \leftarrow r.parent.modifiedFiles$
if C_x **then**
  $result \leftarrow result \wedge (inputs(y) \cap S_{mod} \neq \emptyset)$
else
  $result \leftarrow result \wedge (inputs(y) \cap S_{mod} = \emptyset)$
end
if C_z **then**
  $result \leftarrow result \wedge (outputs(y) \cap S_{mod} \neq \emptyset)$
else
  $result \leftarrow result \wedge (outputs(y) \cap S_{mod} = \emptyset)$
end
return *result*

Algorithm 3: Checks if a given `Revision` is a semantic transformation that adheres to a provided configuration.

Data: A revision r, and a configuration $Conf = \langle C_x, C_y, C_z \rangle$
Result: true iff r fulfills the role of syntactic definition and adheres to $Conf$

if $\neg(r$ *instanceof* `MetaModelRevision`$)$ **then**
  return *false*
end
$S_{curr} \leftarrow r.parent$ **if** C_y **then**
  $Q \leftarrow S_{curr}.modifiedFiles$
else
  $Q \leftarrow S_{curr}.prev.allFiles$
end
if C_x **then**
  $result \leftarrow result \wedge \exists_{y:\text{TransformationRevision}} [z \in output(y) \wedge y \in Q \wedge input(y) \cap S_{curr}.modifiedfiles \neq \emptyset]$
else
  $result \leftarrow result \wedge \exists_{y:\text{TransformationRevision}} [z \in input(y) \wedge y \in Q \wedge input(y) \neq \emptyset \wedge input(y) \cap S_{curr}.modifiedfiles = \emptyset]$
end
return *result*

Algorithm 4: Checks if a given `Revision` is a semantic domain definition that adheres to a provided configuration.