How MetaEdit+ Supports Co-Evolution of Modeling Languages, Tools and Models

Steven Kelly
MetaCase
Jyväskylä, Finland
stevek@metacase.com
ORCID: 0000-0003-0931-157X

Juha-Pekka Tolvanen
MetaCase
Jyväskylä, Finland
jpt@metacase.com
ORCID: 0000-0002-6409-5972

***Abstract*—Domain-specific modeling languages need to evolve when the domain or development needs change, and this leads to a need for co-evolution of related artifacts. We demonstrate how MetaEdit+, a mature commercial language workbench, supports co-evolution of domain-specific modeling languages, tools and models. The demonstration is broken down into 12 different co-evolution scenarios, showing how tools and models update in sync with language changes. In all scenarios of language evolution MetaEdit+ editors open and enable working with existing models, and the models are typically automatically updated without the need to create migration or model transformation programs. When automatic co-evolution is not possible MetaEdit+ points to the items requiring intervention.**

Keywords—domain-specific modeling, co-evolution, domain-specific language, maintenance, metamodel evolution, model evolution

I. Introduction

Domain-specific languages need to evolve when the domain or development needs change. This calls for tool support that enables and supports evolution by co-evolving artifacts that depend on the language, like tools and existing models. However, most work discussing and evaluating tools for domain-specific languages, also called language workbenches, has focused on the initial language creation phase (e.g. [1][2][3]) rather than on language refinement and evolution in the maintenance phase. In practice, the maintenance phase can be seen as the most significant as it tends to require the most resources: as well as tool co-evolution, it requires co-evolution of potentially many models made with the language.

Research on co-evolution has focused on co-evolution of models alongside language abstract syntax (for a survey see [4]) with less work on full language definitions including constraints and concrete syntax, and their evolution. There is also a lack of research covering tools' capabilities to support co-evolution. While work has been done to study limitations of Eclipse-based editors, like GMF [5] or Sirius [6], less has been done on evaluating commercial tools applied in industry.

In this paper we follow a recent evaluation framework [7] to demonstrate how MetaEdit+ [8] supports co-evolution of graphical modeling languages and models. The demonstration shows that unlike tools in [5] and [6], none of the co-evolution situations breaks MetaEdit+ tools. All models open with the updated language allowing modelers to continue their work – and if automated co-evolution is not possible MetaEdit+ points to the model elements requiring human intervention.

II. Co-evolution Evaluation Framework

We demonstrate co-evolution capabilities of MetaEdit+ across the four aspects of [7], two covering the change itself and two covering its results. The first aspect is the location of the change: metamodel (abstract syntax), constraints, and concrete syntax defining the notation or representation. The second is the nature of the change: adding, renaming, removing or changing parts of the language definition. These two aspects are presented in Table I, giving 12 different co-evolution scenarios.

The third aspect is the location adversely impacted by the change: other parts of the language definition, the tool support for modeling, the generators, or existing models. The fourth aspect is the tool's ability to support the given co-evolution scenario, scored from one to five:

1. When creating a new artifact, the editor does not open, or gives tool errors or warnings.
2. Editor opens for creating a new artifact but does not provide the functionality expected.
3. Editor allows creating a new artifact but support for viewing and editing earlier artifacts is incomplete.
4. Editor opens and asks for human intervention to finalize co-evolution of earlier artifacts.
   (4½ if existing models behave and generate correctly, and deprecation guidance is provided where needed.)
5. Editor opens with fully co-evolved earlier artifacts.

III. Evaluation Method

We conduct the demonstration by implementing each of the 12 scenarios by refining the Gothic Security modeling language presented in [9] (state-based modeling of the secret doors and revolving bookcases of spy films: see Fig. 4 for an example).

TABLE I. Location of Change vs. Nature of Change [7]

Location of Change ↓	*Nature of Change*			
	Add	**Rename**	**Remove**	**Change**
Metamodel	1	4	7	10
Constraints	2	5	8	11
Notation	3	6	9	12

For each scenario from Table I, there is a concrete task to demonstrate and be evaluated, taken from [7]:

1. Add element to metamodel: Add a new Reset element to State machine, with a set of events that trigger it.
2. Add constraint: Only one Reset can be defined in a State machine, and it can connect to only one State there.
3. Add notation: The symbol for Reset is created.
4. Rename element in metamodel: State is renamed to Situation.
5. Rename constraint: In MetaEdit+, constraints do not have names, so no change is needed.
6. Rename notation: The symbol for Situation is renamed.
7. Remove element from metamodel: The Reset element is removed from State machine.
8. Remove constraint: Reset is not allowed to have a relationship to Situation.
9. Remove notation: Reset's symbol is removed.
10. Change metamodel: The Transition relationship's Trigger property is moved to the Source role (=relationship start).
11. Change constraint: Add Start, then update old Reset constraints to point to Start instead, and add Start into the original Transition binding.
12. Change notation: Make the Situation symbol refer to a different library symbol.

Others can repeat these scenarios and thus perform the same steps by downloading the MetaEdit+ repository from https://github.com/mccjpt/Gothic and following the instructions there. The initial version is the starting point to repeat this demonstration and tool evaluation. The last version includes the status with all scenarios implemented. In addition to Git's textual version history, the MetaEdit+ repository itself also provides the change history of the models for all 12 scenarios, available from the Changes & Versions Tool of MetaEdit+. These version history features of MetaEdit+ are built-in, available for all modeling languages and all kinds of changes, both in languages and models.

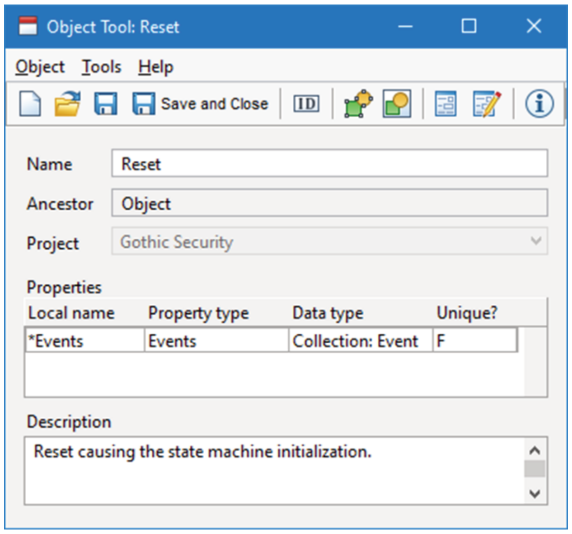
IV. Tool Evaluated: MetaEdit+

MetaEdit+ [8][10] is a mature language workbench that supports diagram, matrix and table representations. It enables collaborative work on both levels: Multiple people can edit the same language definition and multiple people can edit the models at the same time. MetaEdit+ can be used as single or multi-user local installations or remotely in the cloud. MetaEdit+ is commercially successful, used in both industry and academia, and is available to download at https://metacase.com.

V. Demonstrating Co-Evolution Scenarios

A. Adding New Language Elements: scenarios 1–3

1. Add a new Reset element to State machine, with a set of events that trigger it. A new object type called 'Reset' is defined and added to the language with the Graph Tool. It has one property type called 'Events' as in Fig. 1. This property contains a set of events and refers to the 'Event' type that is already defined in the metamodel as part of Transitions.



Name: Reset
Ancestor: Object
Project: Gothic Security

Properties

Local name	Property type	Data type	Unique?
*Events	Events	Collection: Event	F

Description: Reset causing the state machine initialization.

Fig. 1 Adding 'Reset' to metamodel.

After the change, editors are automatically updated and Reset elements can be added. Editors show a default notation for 'Reset' – a proper notation is added later in scenario #3.

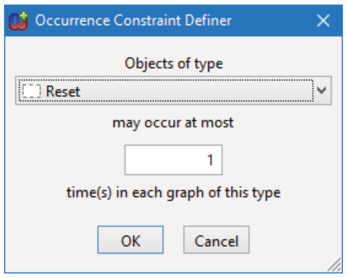
2. Add a constraint that only one Reset can be defined, and it can connect to only one State. The Graph Tool is used to add three things: an occurrence constraint to limit the number of Resets in a State machine to one (Fig. 2), a binding to allow a 'Transition' from 'Reset' to 'State', and a connectivity constraint to only allow each 'Reset' to be in one 'Transition'.

After adding these constraints, existing models may no longer follow the language definition: modelers may have created multiple Resets earlier. To help users update models, the language engineer defines a model annotation or checking report to warn where there are multiple Resets. In our case, we make a checking report that show errors at modeling time at the bottom of the Diagram Editor, as shown later in Fig. 4.

When the constraint is added, all existing models still open and editors update automatically. If a graph contains multiple Resets, the checking report guides the modeler to leave just one.

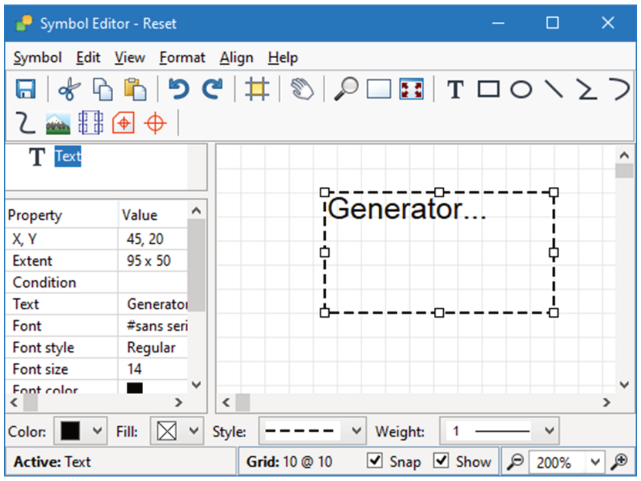
3. Add a notation symbol for Reset. The Symbol Editor of MetaEdit+ is used to define the Reset notation used in [9]: a dotted rectangle listing the reset events. Fig. 3 shows this definition.

After this change, editors update automatically, and models show the new symbol for all Resets. The symbol is also shown elsewhere in the user interface, e.g. in the toolbar, tree views and



Objects of type: Reset
may occur at most 1 time(s) in each graph of this type

Fig. 2. Setting occurrence constraint for 'Reset'.



Property	Value
X, Y	45, 20
Extent	95 x 50
Condition	
Text	Generator
Font	#sans seri
Font style	Regular
Font size	14
Font color	■

Active: Text | Grid: 10 @ 10 | ☑ Snap | ☑ Show | 200%

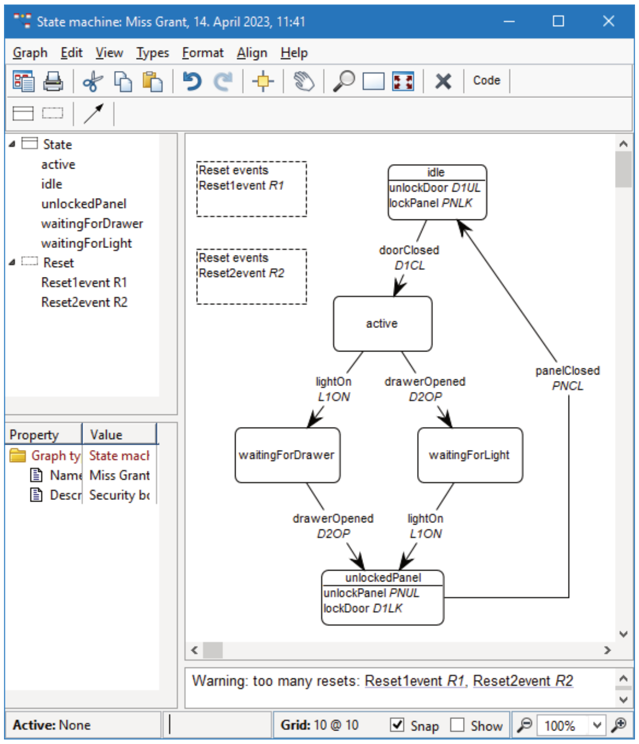
Fig. 3. Adding symbol for ‘Reset’

browsers (as shown in Fig. 4). If the symbol is hard to read when scaled to fit a toolbar button, the language engineer can define a dedicated icon with the Icon Editor of MetaEdit+.

B. *Renaming Language Elements*

4. Rename State to Situation in the metamodel. ‘State’ is renamed to ‘Situation’ by changing its name in the Object Tool. This renaming is automatically updated to constraints and editors. The editors update automatically, with all States in models now showing as Situations.

If there is already a generator that explicitly references ‘State’ by name, it must be updated by the language engineer. The change can be made in the Generator Editor using find and replace (wildcards and polymorphism make fully automatic



Property	Value
Graph ty	State macl
Name	Miss Grant
Descr	Security bı

Warning: too many resets: Reset1event *R1*, Reset2event *R2*

Active: None | Grid: 10 @ 10 | ☑ Snap | ☐ Show | 100%

Fig. 4. Model in editor after adding constraint and symbol for ‘Reset’

refactoring hard). If there are also other (different) elements named ‘State’, e.g. in different dialects of state machines, the search can be restricted to a specific modeling language.

5. Rename a constraint. In MetaEdit+, constraints do not have names, so no change is needed.

6. Rename the Situation symbol/notation. Normally, symbols in MetaEdit+ are directly related to language elements and do not have names. A symbol can, however, also be stored by name in a symbol library, and another symbol can incorporate it from the library by referencing it by name in a Template element. For instance, the ‘Situation’ symbol uses the ‘Rectangle’ library symbol. If ‘Rectangle’ is opened from ‘Situation’ and renamed to ‘BlackRectangle’, ‘Situation’ will automatically use the new name. (If there are other symbols also referring to ‘Rectangle’, the language engineer may update them too to refer to ‘BlackRectangle’.) After this update, the new notation is automatically reflected and applied in models.

C. *Removing Language Elements*

7. Remove the Reset element from State machine. The ‘Reset’ element can be removed from the Graph Tool’s Types (see Fig. 5). This removes it from the language — it is no longer possible to add Reset instances. This leaves us with the question of how to treat existing Reset instances in models. Some tools fail to support them anymore, leading to editors giving errors opening existing models. Another option is deleting all existing instances, but that generally loses too much information. MetaEdit+ follows the mainstream approach of programming language and natural language, effectively obsoleting Reset. Existing instances remain, they can be viewed and edited, and they still work and generate code just as before.

This approach of obsoleting rather than hard deletion allows language users to see and update design data, while guiding them not to use the old language concept anymore. The language engineer can implement a model checking report to list instances that should be deleted (or replaced). Alternatively, MetaEdit+ offers an API [11] that can be used to automate the update of models. And if and when language engineers indeed want a full permanent deletion, MetaEdit+ also supports that and checks for existing instances or references from other metamodel elements.

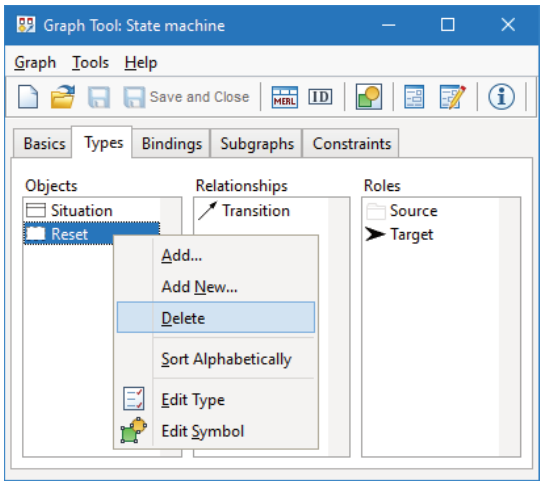


Fig. 5. Removing ‘Reset’ from the metamodel

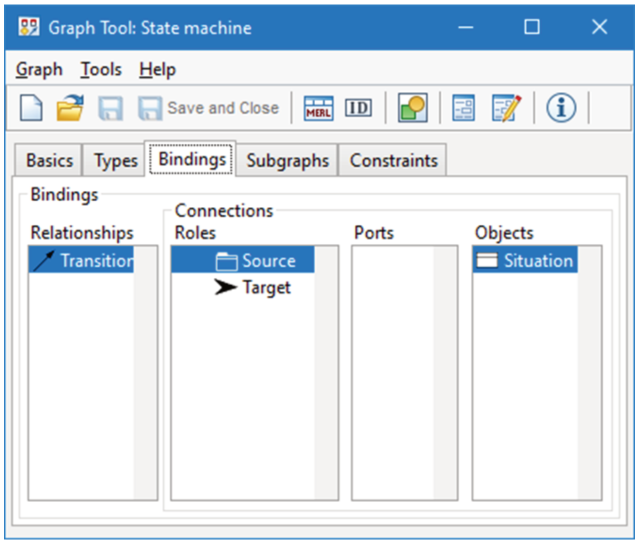


Fig. 6. Removing binding constraint from the language

8. Remove the constraint allowing Reset to have a relationship to Situation. This constraint is the binding created in scenario 2: a positive constraint allowing something, rather than preventing something. It can be removed from the list of bindings in Graph Tool (Fig. 6). Removing this constraint does not require additional actions from the language engineer nor from language users, given the checking report in scenario 7.

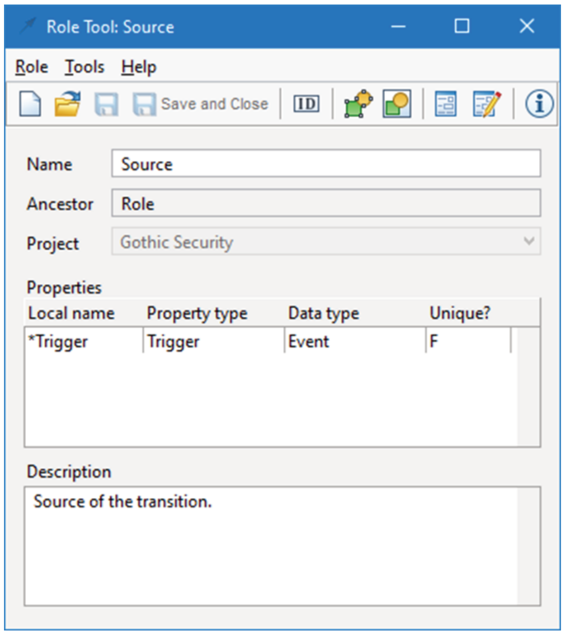
9. Remove notation for Reset. The symbol for a language element, or any part of the symbol definition, can be removed in the Symbol Editor.

After removing metamodel elements, constraints or symbols, editors open existing models and provide the expected modeling functionality. For scenarios 7 and 8 modelers are guided to update named model elements: warnings that Resets are to be removed (which will also remove their relationships). After scenario 9 the defined symbol is no longer shown in the models; if a symbol is needed, like here for old Reset instances, a default symbol is shown.

D. Changing Links on Existing Language Elements

10. Change metamodel by moving Transition's Trigger property to its Source role. In MetaEdit+, changing a reference to an existing element in the metamodel, like moving the 'Trigger' property to the 'Source' role, is based on a direct link rather than an indirect reference by name. Rather than creating a new property type, as in scenario 1, we use the existing 'Trigger' property type in a new property slot in 'Source' (Fig. 7).

After this change the 'Transition' relationship too still has the 'Trigger' property slot, so information is not lost and generators continue to work. Keeping 'Trigger' in 'Transition' is useful for this interim period, allowing current Trigger information to be moved to the 'Source' role. This can be done manually by reusing the existing Trigger event from the Transition in the Source, or automatically by calling the MetaEdit+ API. Language engineers can also prevent creating new 'Triggers' in 'Transitions' by making this property read-



Name	Source
Ancestor	Role
Project	Gothic Security

Properties

Local name	Property type	Data type	Unique?
*Trigger	Trigger	Event	F

Description

Source of the transition.

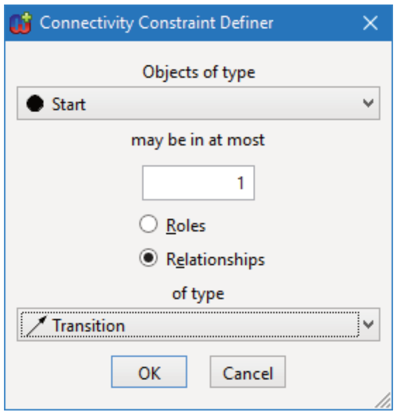
Fig. 7. Changing the 'Trigger' property to be in the 'Source' role.

only. Language engineers can also make an annotation or report as in scenario 2 to highlight the change policy wanted.

For the interim period during manual updates, the symbol in 'Transition' can be moved to 'Source' and updated to show the Trigger from Source or, if missing, that from Transition. If they do not match, an error symbol can be shown.

11. Change constraint: Add Start, then update old Reset constraints to point to Start instead, and add Start into the original Transition binding. First a new object type ('Start') is added in a similar way to scenario 1. Next the existing constraints set in scenario 2 for 'Reset' are updated, changing 'Reset' to 'Start'. Fig. 8. shows a connectivity constraint changed from 'Reset' to 'Start' allowing to have only one Transition. To finalize the scenario, 'Start' is added to the existing binding constraint in Graph Tool by including it alongside 'Situation' in the objects for the 'Source' role.

12. Change Situation's notation to refer to a different library symbol. In the Symbol Editor for 'Situation', the



Objects of type
Start
may be in at most
1
( ) Roles
(●) Relationships
of type
Transition

Fig. 8. Changing connectivity constraint for 'Start'.

template element that uses library symbol ‘BlueRectangle’ is changed to use library symbol ‘Situation’.

During the evolution through these changes, editors continue to work without errors or omissions, and old models open automatically. For scenario 10, modelers cannot add trigger information to transitions anymore and they see notifications to update the models. If model transformation is used for 10 and 11, existing models also update automatically, moving Trigger information to the Source role.

VI. Evaluation Summary

Table II summarizes the results of the evaluation by showing the score for each of the 12 scenarios. The overall score (bold) for each scenario is given first, above sub-scores for affected areas (italic). In most cases, the overall score is the highest, 5: the editor opens with fully co-evolved earlier artifacts. In no scenario is there an adverse impact on the metamodel, constraints or notation, nor on the tool functionality: co-evolution sub-scores for these are all 5. Moreover, none of the 12 co-evolution scenarios broke the other tooling functionality beyond editors, as browsers, model management tools, collaboration, change history, versioning etc. continue to work.

The situations where language users would need to act are cases with score 4½. In all these cases only the model must be updated, e.g. in scenario 2 when constraints are not met and a single Reset must be chosen. Here the modeler sees a notification calling for a human decision. Other scenarios with score 4½ can be handled similarly — or the language engineer can update the models or use the MetaEdit+ API to automate the update. When the API is used, the score would be 5 when removing Reset from the metamodel (scenario 7), removing the binding constraint (scenario 8), or changing the metamodel links to the Event property (scenario 10). In scenario 4, the renaming of an element in the metamodel requires a manual find and replace to update the generators, so this has a co-evolution score of 4. This update is performed by the language engineer and does not call for action from language users.

VII. Conclusions

We demonstrated co-evolution of modeling languages and models in MetaEdit+, following the evaluation framework, example language and model, evolution scenarios and scoring in [7]. The detailed evaluation shows that editors of MetaEdit+ do not break in any of the cases of co-evolution. More importantly, existing models always open in MetaEdit+ after language changes and the language and models change history can be viewed. These have proved to be vital features of MetaEdit+ in industrial use, since its first version in 1995.

While here a single person conducted the language engineering and modeling, in a single language and a single model, the MetaEdit+ principles shown scale to industrial use too. First, models also update for all other users working collaboratively in that MetaEdit+ repository: they will see the results at the start of their next transaction, in both the models and the editor itself, with no need to exit or explicitly update. Multiple people can also work on the language definition at the same time. Secondly, the co-evolution happens similarly regardless of how many languages there are. Thirdly, the upgrade mechanisms are robust with respect to skipping intermediate language versions and updating straight to the most recent version, or deferring manual updates over versions. Finally, the co-evolution of models works the same for large models having hundreds of thousands of elements, without delays or having to explicitly process models separately.

Others can repeat this demonstration and validate its results by downloading the material from GitHub. The evaluation framework will also hopefully be applied to other tools to assess and compare their co-evolution support.

Table II. MetaEdit+ Co-Evolution Evaluation Scores: *metamodel, constraints, notation* | *generator, tool, model* [7]

Location of Change ↓	***Nature of Change***			
	Add	**Rename**	**Remove**	**Change**
Metamodel	**5** *555|555*	**4** *555|455*	**4½** *555|554½*	**4½** *555|554½*
Constraints	**4½** *555|554½*	—	**4½** *555|554½*	**5** *555|555*
Notation	**5** *555|555*	**5** *555|555*	**5** *555|555*	**5** *555|555*

References

[1] S. Erdweg, T. van der Storm, M. Voelter, M. Boersma, R. Bosman, W.R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P.J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth and J. van der Woning. “The state of the art in language workbenches”, *Software Language Engineering (SLE 2013),* LNCS, vol. 8225. Springer, Cham. 2013.

[2] A. El Kouhen, C. Dumoulin, S. Gérard and P. Boulet, “Evaluation of modelling tools adaptation”, CNRS HAL hal-00706701, 2012. http://tinyurl.com/gerard12

[3] J.-P. Tolvanen and S. Kelly, “Effort used to create Domain-Specific Modeling languages”, *ACM/IEEE 21st Int. Conf. Model Driven Engineering Languages and Systems (MoDELS 2018)*, ACM, 2018. https://doi.org/10.1145/3239372.3239410

[4] Hebig, R., Khelladi, D. and Bendraou, R., “Approaches to co-evolution of metamodels and models: A survey”, *IEEE Transactions on Software Engineering*, vol. 43, no. 5, May 1, 2017

[5] D. Di Ruscio, R. Lämmel and A. Pierantonio, “Automated co-evolution of GMF editor models”, *Int. Conf. Software Language Engineering (SLE 2010)*, pp. 143–162, Springer, 2010.

[6] A. Pierantonio, J. Di Rocco, D. Di Ruscio and H. Narayanankutty, “Resilience in Sirius editors: Understanding the impact of metamodel changes”, *ACM/IEEE Int. Conf. Model Driven Engineering Languages and Systems (MoDELS 2018)*, 2018.

[7] J.-P. Tolvanen and S. Kelly, “Evaluating tool support for co-evolution of modeling languages, tools and models”, *26th Int. Conf. Model Driven Engineering Languages & Systems (MoDELS 2023): Companion Proceedings, Workshop on Models and Evolution*, ACM, 2023.

[8] S. Kelly, K. Lyytinen and M. Rossi, “MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment”, *Conf. Advanced Information Systems Engineering (CAiSE 1996)*, 1996.

[9] M. Fowler, *Domain-Specific Languages*, Addison-Wesley, 2010.

[10] MetaCase, “MetaEdit+ - User’s Guides” version 5.5, https://metacase.com/support/55/manuals (accessed July 20, 2023)

[11] MetaCase, “MetaEdit+ Workbench User’s Guide” version 5.5, Chapter 9, https://metacase.com/support/55/manuals/mwb/Mw.html#Mw-9.html (accessed July 20, 2023)