Software and Systems Modeling
https://doi.org/10.1007/s10270-024-01218-5

THEME SECTION PAPER

A framework for evaluating tool support for co-evolution of modeling languages, tools and models

Juha-Pekka Tolvanen[1,2] · Steven Kelly[1] · Juri Di Rocco[3] · Alfonso Pierantonio[3] · Giordano Tinella[3]

Received: 2 February 2024 / Revised: 22 June 2024 / Accepted: 5 September 2024

Abstract
We present a framework for evaluating language workbenches' capabilities for co-evolution of graphical modeling languages, modeling tools and models. As with programming, maintenance tasks such as language refinement and enhancement typically account for more work than the initial development phase. Modeling languages have the added challenge of keeping tools and existing models in step with the evolving language. As domain-specific modeling languages and tools have started to be used widely, thanks to reports of significant productivity improvements, some language workbench users have indeed reported problems with co-evolution of tools and models. Our tool-agnostic evaluation framework aims to cover changes across the whole language definition: the abstract syntax, concrete syntax, and constraints. Change impact is assessed for knock-on effects within the language definition, the modeling tools, semantics via generators, and existing models. We demonstrate the viability of the framework by evaluating MetaEdit+, EMF/Sirius and Jjodel, providing a detailed evaluation process for others to repeat with their tools. The results of the evaluation show differences among the tools: from editors not opening correctly or at all, through highlighting of items requiring manual intervention, to fully automatic updates of languages, models and editors. We call for industry to evaluate their tool choices with the framework, tool developers to extend their tool support for co-evolution, and researchers to refine the evaluation framework and evaluations presented.

Keywords Domain-specific modeling · Domain-specific language · Evolution · Maintenance · Metamodel evolution · Model evolution

1 Introduction

In software and systems development, refinement, enhancement and other maintenance tasks normally account for more work than the initial development phase. This applies to domain-specific languages and models too, including their co-evolution. Compared to general purpose languages, domain-specific languages (DSLs) and domain-specific modeling (DSM) languages evolve more frequently – following changes in the domain and in the development needs [1–3]. A recent DSL practitioner survey [4] found that 86% of respondents reported language evolution, and recommended considering evolution as an intrinsic part of DSL creation.

An important characteristic of language evolution is that changes must be reflected in artifacts already made with the language: we want to preserve that work and move artifacts to the new language version. This enables sustainable development, both of the applications made by modeling, and of the language itself. The economic and technical benefits of DSM co-evolution are clear, but there are also important benefits for other aspects of sustainability [5]. Poor co-evolution support can lead to language stagnation [2], harming communication and social sustainability as the distance between the stagnant language and its evolving domain increases. As the

Communicated by Dalila Tamzalit and Ludovico Iovino.

✉ Juha-Pekka Tolvanen
jpt@metacase.com

Steven Kelly
stevek@metacase.com

Juri Di Rocco
juri.dirocco@univaq.it

Alfonso Pierantonio
alfonso.pierantonio@univaq.it

Giordano Tinella
giordano.tinella@student.univaq.it

1 MetaCase, FI-40500 Jyväskylä, Finland

2 University of Jyväskylä, FI-40014 Jyväskylä, Finland

3 Università degli Studi dell'Aquila, I-67100 L'Aquila, Italy

Published online: 10 October 2024

gulf widens, the 5–10 times productivity increase [1] offered by DSM falls, leading to increased resource usage in development [6]. With 'software engineer' being one of the largest job categories these days, and IT equipment a significant consumer of energy, even environmental sustainability is at risk. Conversely, enabling co-evolution maintains the high sustainability benefits of DSM, from developer productivity to the ease of targeting new, lower-energy platforms with minimal effort through new generators.

The co-evolution of a domain-specific modeling language has an important characteristic due to its restricted use. If the language is made to address a narrow domain within a single company or its team, as reported in over a hundred cases [7], then it is likely that all language users are known, their specifications made with the language can be accessed, and data to assess the impact of language evolution can be inspected in all the language use contexts. Conversely, the number of users is significantly smaller than for general purpose languages, so the effort that can sensibly be spent per language change is smaller.

Despite the importance of co-evolution, most research on tools for DSLs and DSM, also called language workbenches [8, 9], has focused on the initial steps of creating the language (e.g., [9–11]), rather than the refinement and maintenance of languages and models. Research on co-evolution has focused on changes in certain parts of a language—such as in its metamodel or transformations—but not covered all aspects of a language together, as we aim to in this paper.

In the current paper, we propose an evaluation framework for assessing how graphical modeling tools respond to language changes in terms of co-evolution. This paper expands on previous evaluations conducted within single tools [12–14] by introducing a tool-agnostic framework designed to assess the co-evolution capabilities of language workbenches. The evaluation now covers various changes across the whole language definition: the abstract syntax, concrete syntax, and constraints. Change impact is assessed for knock-on effects across the whole tool: within the language definition, the modeling tools, semantics via generators, and existing models. To ensure applicable results, a series of realistic change scenarios are provided. The impact assessment produces a score for each impact location for each scenario, giving a broad and detailed evaluation of tool co-evolution performance. The evaluation framework addresses co-evolution on an industrial scale by taking into account the amount of work needed and the effects of large numbers and sizes of models. To show its viability, we apply the evaluation framework to three graphical language workbenches: MetaEdit+ [15], EMF/Sirius [16] and Jjodel [17]. The selected tools vary by their technologies, history, conceptual organization of the development process, features provided, and the characteristics of the modeling environment.

Detailed material is provided for others to repeat the evaluation process to validate it and to evaluate other modeling tools. Our work allows industry to evaluate their tool choices for long term usage, and tool developers to investigate and extend their tool's support for co-evolution.

We start by describing previous research on language, model and tool co-evolution (Sect. 2), and try applying an existing evaluation framework (Sect. 3). This leads us to suggest the set of aspects to include in our own framework presented in Sect. 4. Section 5 presents the procedure for applying our evaluation framework: an example language and model, along with a set of evolutionary steps to test all the aspects of co-evolution. Section 6 presents the tools evaluated and in Sect. 7 our framework is then applied by following its procedure to evaluate the co-evolution support in the tools. In Sect. 8 we discuss the results of tool evaluations and assess the evaluation framework based on these experiences. In Sect. 9 we conclude with proposed directions for future extension and verification.

2 Research on co-evolution with tools

Research on co-evolution has focused on metamodels and models, with less research inspecting co-evolution of tool support, and mostly only experience reports mentioning both. Most of the time, the proposed approaches focus only on a specific kind of artifact, forcing the language engineer to become fluent with too many different techniques, e.g., restoring the validity of models requires techniques that cannot be used to restore the consistency of transformations and vice versa. Consequently, the co-evolution of entire modeling environments is an intrinsically complex task. It might lead to lock-in scenarios in which the modeler prefers to leave the metamodel unchanged, to avoid facing the difficulties of propagating the changes throughout the ecosystem (e.g., see [2, 18]).

In the rest of the section, we analyze existing works on the co-evolution of metamodels and modeling artifacts dependent upon them, and then consider co-evolution of metamodels and modeling tools.

2.1 Research on language and model co-evolution

The large body of work on co-evolution has mainly focused on co-evolution of metamodels and models, and only partly considering evolution in other parts of the language definition, such as its constraints, notation or generators/transformations. A prevailing approach [19] has been to create transformations acting upon models (e.g., [20–23]) to enable their co-evolution with metamodel changes. Once defined, an appropriate transformation would be executed each time the language evolves. See [3] for a survey of meta-

model and model co-evolution approaches. Note that while this survey covers a wide range of approaches, it is based on a literature survey rather than the co-evolution approaches applied in current commonly used tools. Moreover, as many of the surveyed approaches are ongoing work, Hebig [3] concludes that there is little data for determining their applicability in industrial contexts.

In [3, 19], one class of approaches suggests the use of transformation languages to co-evolve models each time the metamodel changes ([24]). The transformations are made for each case and can be partly automatically produced, e.g., starting from metamodel differencing. A second class of approaches is based on identifying predefined co-evolution strategies or allowing users to specify them ([25–27]); typically, the evolution of the metamodel and models progresses in parallel. A third approach is searching based on model data, not metamodel, to co-evolve the model to the new metamodel ([28, 29]). The final approach identified was labeled as identifying complex metamodel changes.

While these approaches cover co-evolution, they focus on changes in models—although Hebig [3] recognizes that evolution also requires the co-evolution of other artifacts such as transformations or constraints. We aim to inspect all aspects of modeling language change with our evaluation framework.

Co-evolution of transformations and generators has been a less popular research subject, and only a few works have been presented in the last decade or so (see, e.g., [30–32]); similarly for the co-evolution of constraints [33–35]. Some likely reasons are that it is considered as a normal language engineering task and does not have such clear implications for the work of modelers. Also, the wide variety of changes that are possible makes it less automatable, although tools could provide some support.

The evolution of concrete syntax is also recognized as a language evolution issue, and often involves co-evolution: how the mapping between the abstract and concrete syntax is maintained. While it is recognized in evaluation frameworks like [19], its effect on existing models seems to be strongly dependent on the particular tooling used.

Interestingly, regardless of the kind of artifact to adapt, most approaches are deterministic, having either a specific built-in heuristic or a programmatic definition. However, there is often more than one possible way to update, and no way for the language engineer to decide which way to use in which situations. For instance, when the legal number of instances of a concept in the metamodel is decreased, existing models with more instances would require some to be deleted, with no acceptable way to specify in advance which to keep and which to delete. In [36], the problem is mitigated by proposing an interactive approach, whereas in [37] the related uncertainty problem is addressed by means of a variability model. A different view on this variability problem is put forward in [38], in which the modeling ecosystem is viewed as a megamodel [39] consisting of all artifacts related to a metamodel, and the consistency restoration is executed according to an orientation model, that is a way of selecting exactly one update policy among all potential valid ones.

2.2 Research on language and modeling tool co-evolution

There is relatively little research on how modeling tool support co-evolves alongside the language supported [13]. Publications comparing language workbenches tend to focus on initial language creation phases (e.g., [9–11]) and do not address language evolution, nor the required co-evolution of modeling tool support and existing models. This is somewhat surprising: in a study [40] on practitioners' modeling challenges, over 60% named evolution of language as a challenge in tools. This need for co-evolution also exists in fixed language modeling tools, when either the language changes or the metamodel of the language is refactored significantly. For example, after moving from SysML 1.6 to SysML 2.0 one of the key concepts, 'Block', no longer exists [41].

Another recent study [42], focusing directly on DSM and DSL tools, indicated that a tool's ability to update models automatically when the metamodel changes is considered the second most wanted semantic editor feature—the most important being highlighting model elements and associated error messages.

Studies directly evaluating tools take a wider view of languages than just metamodels, as at least the editor functionality is inevitably visible, and support for co-evolution quickly becomes visible even with just incremental language definition. In studies evaluating the capabilities of Eclipse-based editors [12, 13], concrete syntax is also recognized as a part of the language definition. GMF-based tools are found to lack co-evolution support in many ways [12], and Sirius-based editors break or are incomplete in several co-evolution situations [13] (see Table 1 in Sect. 3 below). Both studies are performed and reported in a methodologically rigorous way, allowing others to repeat and validate them. These studies are however restricted in the sense that they do not report changes that deal with constraints or generators/transformations related to the language definitions.

Evaluations [12, 13, 23] also vary in their classification of change impact:

1. Non-breaking: the editor can open [13] or models conform to the metamodel [23];
2. Complete: all metamodel elements have graphical counterparts in the editor [13];
3. Valid: the editor exposes correct behavior, e.g., one can create a model conforming to the new metamodel [13])
4. Resolvable: an automatic procedure can restore validity and completeness after a breaking change [13, 23]; '3

sound' in [12] is similar, but others there do not map well.

Less research seems to have evaluated commercial tools on an industrial scale. What is industry-scale may of course vary, but we expect models to be large (≥10,000 elements), have many language users (several, dozens or hundreds rather than one or a few), and languages to evolve and be used over a long period of time (over a decade). Over the years, the tools applied also evolve with new versions, sometimes leading to users losing work [6].

Reports on industrial use provide another source for inspecting co-evolution—often related to a specific tool. At Philips, language engineers updated instances manually each time the grammar in Xtext changed [43]. This was recognized as a limitation, but was considered feasible for their case as the number and size of models was small. Since manual processes become tedious, error-prone and costly with larger models, automated solutions are considered mandatory. At the opposite end of the size range, a study at ASML [44] with over 5000 models based on 22 different metamodels claims that 20% of time was spent on maintenance effort, calling for automated support.

In [25], a GMF language had 214 changes from version 1.0 to 2.0; a hand-written migrator was provided for language definitions, but not for models made from them nor for the significant amounts of custom code commonly added to build a GMF-based editor. Another Xtext case, ([45] p. 263) implemented a generator to automate transformations that could run over many models in a batch. The actual mappings between two metamodels were made manually. A transformation-based approach was also applied with Microsoft DSL tools, for which Avanade presented a mapping language as a basis for generating model converters [46]. A report [47] by Siemens indicated that migration scripts were needed to keep existing models working with MPS. Applying them was challenging because users also had their own branches of models. When the language and generators changed, it became hard to maintain tool support, so finally they hosted custom RCP instances of MPS, one per language version, matching each model release branch.

3 Exploratory evaluation and pilot

Before proposing a new framework, one should look at existing frameworks. Particular value is to be found in going beyond a simple mention as part of related research, and actually applying an existing framework in an evaluation. This helps to build on existing work, recognizing its strengths and identifying areas where it could be extended.

Table 1 Summary of metamodel change impact in tools (color figure online)

	GMF [12]	Sirius [13]	MetaEdit+
1. add concrete class	x	x	o
2. add abstract class	x	o	o
3. insert superclass	o	x	o
4. delete class	x	x	o
5. rename class	x	x	o
6. add property	x	xo	o
7. delete property	x	x	o
8. rename property	x	x	o
9. move property	x	x	o
10. pull up property	x	o	o
11. change property type	x	xo	o

3.1 Exploratory evaluation with existing framework

Di Ruscio et al. [12] applied a set of criteria to language and tool co-evolution in GMF, and Pierantonio et al. [13] used this existing set to evaluate Sirius. Parts of the framework there are also used in other research mentioned earlier. Use of a common benchmark in this way is a good example of increasing maturity—even if we also want to continue and improve the benchmark, e.g., to cover co-evolution of models and generators as well as tools.

As an exploratory step, we thus took the 11 criteria tested on both GMF and Sirius in the evaluations above, and applied them to evaluate MetaEdit+ (5.5 Build 47). The results are shown in Table 1. The coloring is green and o for full success, red and x otherwise, and where Sirius and MetaEdit+ property tests were performed for both simple attributes and more complex references, orange and xo for full success only on attributes. (Full success is defined here as success on all individual columns in the results tables of the original papers, which focused on editor behavior.)

The 11 criteria of the existing framework were relatively easy to interpret in the context of a different tool and different metametamodel. MetaEdit+ modeling tools handled all the changes well, with no errors or omissions in tool behavior. Although the successful co-evolution results were encouraging, they revealed the need for more in-depth evaluation to identify cases, in MetaEdit+ and other tools, where co-evolution support needs more work.

3.2 New framework and pilot

From the exploratory evaluation, three possible areas of extension could be seen:

- *Location of change*: The existing framework only tested changes to the abstract syntax of the language, and this should be extended to cover changes made to other parts

of the language, such as its concrete syntax, and rules or constraints.
- *Nature of change*: The existing framework only considered certain kinds of change operations, so other kinds of operations should be examined to see if they might raise their own questions of co-evolution.
- *Location impacted by change*: The existing framework only tested the impact of changes on the tooling, and this should be extended to test the impact on other parts of the language definition (partly covered in [13]), the generators and transformations, and existing models.

These three areas were addressed in a new framework, the initial version of which was applied to evaluate MetaEdit+ as a pilot case [14]. In the rest of this paper, we will describe the new framework in its current state, and how it was applied to evaluate a broader set of tools.

4 Framework for evaluating co-evolution support

We separate the co-evolution of languages, models and tools into four aspects. The first aspect is the location of the change, i.e., the part of the modeling language being changed: its abstract syntax, constraints, or concrete syntax. These commonly accepted parts of languages are recognized as evolving by others too (e.g., [19]). Our framework thus takes a wider perspective on co-evolution than addressing purely metamodel-driven co-evolution.

The second aspect is the nature of the change: adding, renaming, removing or changing parts of the language definition. These first two aspects thus concern the change that is made; the remaining two aspects cover the possible (adverse) impact of each change.

The third aspect is the location impacted by the change, i.e., which artifacts are adversely affected by the change: other parts of the language definition, the tool support for modeling, generators, or existing models. As not all changes can be automated without adverse effects, we can also look at the capabilities offered by the tool to support the language designer and/or user through the evolution scenarios.

The fourth aspect is the resulting severity of impact on artifacts, ranging from not opening at all to fully co-evolving. We focus particularly on the user's ability to interact with artifacts via the tools. While tooling too is considered as an artifact in its own right, a tool may work properly but be unable to open a certain model that has become adversely impacted by language evolution; in that case we consider the problem to be in the model artifact rather than the tool as an artifact itself.

4.1 Location of change in language definition

Abstract syntax is typically defined via a metamodel. The metamodel may also express the rules and constraints, or they may be expressed in additional constraint or transformation languages. Using language definitions by OMG as examples, a metamodel in MOF specifies the concepts of the language and the basics of how they are connected, with extra constraints being defined with OCL.[1] In our evaluation framework, we separate these parts accordingly.

Constraints and rules may be strictly enforced or then shown as warnings when violated, e.g., with a red icon in a diagram symbol, or a warning in an error list pane of the editor [48]. If the rule is made as part of the concrete syntax, we will consider it there rather than as a constraint. Similarly if the rule is written as a generator to produce an error list, we will consider it as part of the generator. The deciding factor is thus where a particular tool or language engineer chooses to implement it, rather than whether it is semantically like a constraint.

Concrete syntax defines the notation, making models visible for humans and accessible via the user interface of the tool. Depending on the representation style, a model can be a diagram, matrix, table, tree, text etc. or any of their combinations. Often a concrete syntax is defined for the key metamodel concepts, with the details of models accessible via generic user interfaces such as property sheets.

The semantics of a language may also be subject to change, just like other parts of the language definition. How the semantics is defined varies, depending on factors like the purpose of the language. For example, if used for producing code, the semantics is typically defined via a mapping to a programming language (translational semantics via generation), or models are executed at runtime (interpretative semantics). If the language is mainly targeting communication, sketching or documentation, then the semantics is typically defined in a prose definition of the language and its elements (e.g., as with modeling languages like ArchiMate[2] and SysML[3]).

In any of these three approaches to semantics, a change to the semantics is unlikely to break other parts of the language definition or models—in the same sense of tool errors and omissions as used for other language changes. We thus do not test changes made to the semantics, but we will examine whether changes made elsewhere can have an impact on the parts of the language definition concerned with semantics, e.g., a generator breaking after a language concept is renamed.

[1] https://www.omg.org/spec/OCL/2.4/About-OCL.

[2] https://publications.opengroup.org/standards/archimate.

[3] https://www.omg.org/spec/SysML/1.6.

4.2 Nature of change: add, rename, remove, change

Evolution can happen for example by adding, renaming, removing or changing part of the language definition [23]. Looking more closely we can identify:

- Create + **Add** Reference (e.g., new kind of object in language)
- Change simple content (e.g., number in constraint)
- **Rename**
- **Remove** Reference
- **Change** Reference (e.g., A→B becomes A→C)
- Delete (full deletion)
- Change in hierarchy (e.g., pull up property)
- Change of metatype (e.g., relationship becomes object)
- Change simple datatype (e.g., string becomes int)

We use the word Reference here specifically to mean a link to another first-class concept in the metamodel: e.g., that Use Case diagrams can include Actor objects. The reference can either be direct or by name, with the latter generally being brittle with respect to rename operations, but offering indirection and modularization needed in some cases.

Some of the changes listed are so simple that they should cause no problems in tools or models (e.g., creating a new object type). Others are known to be hard, but familiar from many other branches of software engineering (e.g., a string becomes an int). We will focus on the four changes in bold, which in our experience are the key changes encountered in language evolution [49].

We decided not to include changes that are more in the solution domain of metamodeling (e.g., refactorings of the metamodel, particularly its inheritance hierarchy) rather than its problem domain (what is desired in the language). As seen in the exploratory evaluation, the definition and details of metamodel refactorings are more dependent on the language workbench and metametamodel, and harder to interpret consistently across different tools: not all tools even allow inheritance within a metamodel. Our experience is that refactorings of this kind tend to occur more often at an early stage, before there are enough models to make co-evolution a question. The same results in the language can also normally be achieved by other means, e.g., rather than pulling up a property to a superclass, it can be added to sibling classes: less ideal, but not as serious an issue as not being able to add, rename, remove or change parts of the language itself.

Before moving on from the aspects about the change itself to the aspects covering the possible (adverse) impact of each change, we should consider our practical philosophy for co-evolution. Where a language change reduces the set of valid models, it is rarely a good idea to adopt a strict formalist approach: e.g., deleting parts of models that no longer correspond to the language definition. The deleted parts would contain information and earlier choices that the modeler will often want to see as part of model co-evolution. Since the models have been legal with respect to the earlier language definition, and valid for generation, a more palatable approach is deprecation: allow the old style, but show warnings and guidance on the new style. This can be accompanied by information on how the deprecation will proceed, e.g., initially allowing both old and new, then not allowing creation of further instances of the old style, then showing warnings for the old style, then making the old style fail integrity checks. Particular cases may need more detailed conditions, e.g., only allow generation targeting products that are themselves sunset or in maintenance mode, or change an old property to be read-only or hidden. In most cases the overall aim should be to guide users toward migrating their behavior and existing models to follow the new approach, but there can also be good reasons to allow the old style to remain, e.g., in models that are no longer actively updated, but still in use. When there is a separate reason to update one of those models, it can be updated to the new style first. The use of deprecation for managing co-evolution is well known from programming languages, but also suggested in the specific case of metamodels and models [50, 51].

4.3 Location impacted by change

While we focus here mainly on co-evolution impact on models and modeling tools, a change in one part of the language definition may have an impact on other parts of the language definition. For example, in a typical language engineering task adding a new kind of object to a diagram type generally leads to giving it a graphical symbol as its concrete syntax, adding some rules for it, and updating generators to produce code from it. Similarly removing it from the diagram type may leave no longer needed rules and parts of the generator. In both kinds of cases, we will not consider it a problem if the editors work without errors.

Co-evolution within the language definition is not as significant as co-evolution with models, since it only affects the work of a few language engineers. Also, the size of specifications is smaller in language definitions than in system or software specifications in models. It is nevertheless an important aspect, as limited tool support for language evolution can hinder ongoing refinement, leading to language stagnation.

The most-studied locations impacted by changes in a language definition are the modeling tools and models. As we have discussed these in-depth earlier, there is little to add here. We will just note that by models we refer to the actual model data, not the ability to view or edit it; that will be considered as part of the modeling tool functionality. A hard dividing line may of course be difficult to set.

Finally, the semantics of the language may be adversely impacted by changes elsewhere in the language definition.

Table 2 Location of change versus nature of change

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

As mentioned earlier, we will ignore changes that are simply not made yet: e.g., when adding a new language concept, there will generally be no generation for it, but this is not considered as an error. However, if existing generators now break because of the language change, that is a clear adverse impact.

4.4 Scenarios of co-evolution

Table 2 shows every possible combination of location and nature of change: the scenarios we want to test. While these could be evaluated individually, a coherent sequence of changes gives a more realistic test. We thus order them to form 12 steps or scenarios, in a sequence similar to what we might see in practice. For example, scenario 1 refers to adding an element to the metamodel, and scenario 2 adds a constraint related to the new element.

For each such scenario we evaluate the impact of the change on other parts of the language definition, on the tool's modeling functionality, on generators, and on existing models. We also evaluate how the tool supports the language developer and language user in the change.

4.5 Assessing change impact

Since our focus is on tools' capabilities, the framework is made primarily to evaluate how a given tool can cope with the changes. Tool evaluations [12, 13, 23] characterize tool functionality in a variety of ways, as mentioned earlier. Some of the semantics of those categories seem somewhat unclear, and indeed they seem to be applied somewhat differently in different papers. We will try to follow similar ideas and ordering, but give more concrete descriptions distinguishing the capabilities of editor functionality. Rather than limiting the framework to models, we will use the term 'artifact' to cover the various parts of the language definition or models—either existing artifacts made earlier, or creation of a new artifact of that type in the context of this language. Similarly the term 'editor' covers the parts of the tool used for editing that artifact: a Diagram Editor for a model artifact, a Generator Editor for the generator artifacts, etc. The scoring is:

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 expected functionality.
3 Editor allows creating a new artifact but support for viewing and editing earlier artifacts is incomplete.
3½ Editor opens with complete functionality for new and earlier artifacts but without useful messages on necessary or advisable actions.
4 Editor opens and asks for any necessary human intervention to finalize co-evolution of earlier artifacts.
4½ Editor opens, existing models behave and generate correctly, and deprecation guidance is provided as needed.
5 Editor opens with fully co-evolved earlier artifacts.

Our main focus will be on model artifacts and modeling tool behavior, but we will evaluate for impact on other artifacts too. We will evaluate each of the 12 co-evolution scenarios (nature and location of change), and in each we will give a subscore for each of the six possible locations of impact of change. For example, we will perform a scenario of renaming a language concept, and look whether that had an adverse impact elsewhere in the metamodel, constraints, notation, tooling, generators or models, giving a subscore for each. (More specific descriptions of each score for different kinds of artifacts are available in the Supplementary Material [52].)

Having six subscores for each of 12 scenarios gives 72 data points for each tool. To give a more immediately visible single score for each scenario, a simplified overall score for a scenario will also be shown. The choice of how to calculate the overall score is motivated by four factors. Firstly, our initial investigation showed that in many cases, a tool would lose points on only one subscore in a given scenario. Secondly, the scoring is not necessarily an evenly spaced metric: the difference between a score of 2 and a score of 3 may be different from that between 3 and 4. Taking the mean is thus not strictly speaking statistically appropriate (this is an ordinal scale, so better than a nominal scale, but still not an interval or ratio scale). Thirdly, the overall minimal work needed to repair a poor set of scores in a scenario is, in our experience, generally determined by the worst subscore. Fourthly, the use of the lowest subscore in evaluations of multiple factors is well-attested, particularly in cases where there is a clearly desirable default result of the maximum score—e.g., Euro NCAP car safety testing. We thus choose to show a scenario's overall score as the lowest of its subscores.

The use of scores with halves, such as 3½, is to improve forward and backward compatibility. When a need for a new score value is identified, inserting a new half-value between existing values avoids changing the semantics of existing values. This allows us to maintain better compatibility with the scores from earlier evaluations such as [14].

The scores are given based on the state after performing the operations on the metamodel, constraints or notation:

they measure how well the tool automatically performed any necessary co-evolution. Where further manual operations are needed to complete co-evolution, e.g., to update models, the time for these is included in the effort measurement in the following subsection.

4.6 Effort to perform co-evolution

Research on industrial use [6, 44, 47] and academic studies [25, 26, 51, 53, 54] both find effort and various scalability topics to be relevant in co-evolution. We suggest measuring how much time is spent on the initial change and co-evolution tasks in the language, generators, models and tooling. Time is in many cases directly the variable of interest for work on a language and its models [55]. It is also directly comprehensible by readers without knowledge of the workbench in question. We measure the time for each co-evolution scenario separately (described in Sect. 7.6). Particular attention is paid to times that will increase as model size increases, i.e., co-evolution of models.

We will measure the time from the start of the scenario's changes to the language up to the end of any corrections needed to proceed effectively with modeling work and subsequent scenarios. A single time will be given for each scenario.

5 An example language and model

To make the evaluation concrete and repeatable we use an example from [8]: a state machine for Gothic Security, modeling the secret doors and revolving bookcases of spy films. Figure 1 shows (a) the metamodel as a class diagram, and (b) an example model. These are small enough to be easily implemented in any DSM language workbench, yet large enough to enable conducting all 12 scenarios. The language is a dialect of state machines: states may have commands, and transitions between states have a triggering event. In [8] both commands and events have a name and a code. There are also constraints, evident only from the generated code, e.g., state name is mandatory and unique within the current state machine.

The model shown in Fig. 1b defines the functionality of a system made for a customer called Miss Grant for opening a hidden panel [8]. It also illustrates the concrete syntax of the language. From a model in this language, code can be generated for various targets; when assessing generator co-evolution, we will consider the Java generators.

Following the co-evolution framework presented in Sect. 4 and its Table 2, we have 12 different scenarios to inspect whether making a given change to the language has adverse impacts on other parts of the language, modeling tools, generators or models. For our example language we choose these concrete scenarios:

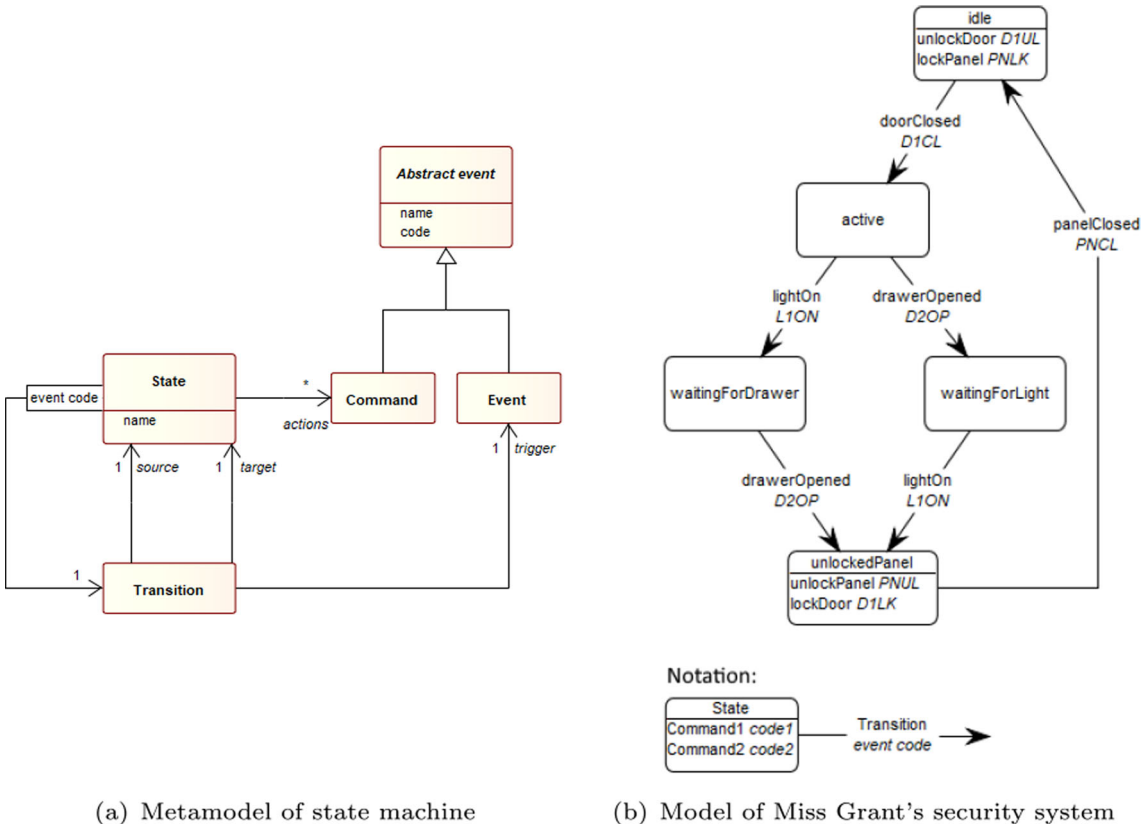
#1 Add concept to metamodel: Add a new Reset element to State machine. A Reset has a set of events that trigger it.
#2 Add constraint: Only one Reset can be defined in a State machine, and it can have only one Transition to a State there.
#3 Add notation: The symbol for Reset is created and associated with Reset.
#4 Rename element in metamodel: State is renamed to Situation.
#5 Rename constraint: Only one Reset can be defined in a State machine.
#6 Rename notation: The symbol used 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 Transition to Situation.
#9 Remove notation: Reset's symbol is removed.
#10 Change metamodel: The Transition's Trigger is moved to be specified in the Source end of the relationship.
#11 Change constraint: Add Start, then update old Reset constraints to apply to Start instead, and allow Start to be the Source of a Transition to a Situation.
#12 Change notation: Make the Situation symbol refer to a different symbol.

These scenarios are generic to state machines and any language workbench should have functionality to define these. Other concrete scenarios would be possible, but these 12 steps are defined so that they can be implemented following each other. In this sense there are 12 sequential versions. All the suggested changes are also evolutionary and not revolutionary: If the language were to change completely, it would be more the case that language engineers would create a new language.

The example model in Fig. 1b is small: one diagram with 5 states, 6 transitions and 21 property values like state names, command codes etc. To address the industrial concerns and to estimate the effort of co-evolution, we make a larger model set by duplication. First we extend this small model by replicating its contents within the same diagram to be 8 times larger: 40 states, 48 transitions and 168 properties. This larger single diagram is then replicated again to 72 diagrams, giving a model set that contains in total 2880 states, 3456 transitions and 12096 properties values.

6 Tools evaluated

In this section, a brief description is given of the graphical language workbench tools evaluated: MetaEdit+, EMF/Sirius and Jjodel. The tools had to support the areas present in the framework (e.g. graphical notation, constraints, genera-



(a) Metamodel of state machine (b) Model of Miss Grant's security system

Fig. 1 Metamodel of State machine and a model based on this state machine (color figure online)

tors), ruling out text-based language workbenches like Xtext[4] or purely graphical drawing tools like Visio[5], which would both be inherently different in terms of their language definitions and co-evolution challenges. The selected tools cover a broad range in many respects, including user-visible factors such as features provided, language editing facilities, and the characteristics of the generated modeling environment; technical factors such as adopted technologies, architecture and persistence formats; and broader factors such as history, funding, the conceptual organization of the development process, maturity and adoption.

- MetaEdit+ is a commercial tool introduced in 1995, with thousands of developers using it world-wide in industry – and academia, as the most commonly used commercial tool in research articles.
- Sirius is an open-source tool released in 2013 based on the Eclipse Modeling Framework (EMF), with both being part of the Eclipse ecosystem, and widely used in many academic and industrial projects.
- Jjodel is a recent academic project (2021) with the intent to overcome certain difficulties, including the accidental complexity of the current open-source tools; moreover, it is a reflective tool that adopts a modern technology stack aligned with the most up-to-date standards.

The comparison could have considered other tools; three seemed a sensible number after an initial pilot with one tool, and the selected tools assure reasonable coverage and diversity within the framework's focus on tools devoted to graphical modeling languages. We invite the many other language workbenches that support graphical languages to try out their tools against the evaluation framework and report their results.

6.1 MetaEdit+

MetaEdit+ [15, 56] is a mature language workbench that supports graphical diagram, matrix and table representations. It enables collaborative work on both language engineering and language use: Multiple people can edit the same language definition and multiple people can use the language at the same time. MetaEdit+ can be used as local installations or remotely in the cloud [57]. MetaEdit+ is commercially successful, used by customers in both industry and academia [58] and is available to download at metacase.com. Support for co-evolution was one requirement for developing MetaEdit+

[4] https://eclipse.dev/Xtext/.

[5] https://microsoft.com/visio/.

[59] and over the years, MetaEdit+ tool version updates have upgraded any languages, generators and models since the first release of version 2.0 in 1995, with fully automatic upgrades since version 3.0 in 1999—covering many tool releases and significant updates of its GOPPRR metametamodel [56].

Language development and maintenance can be carried out in MetaEdit+ in three different ways. The primary way, used here (see Fig. 2), is to use the integrated metamodeling tools in MetaEdit+ Workbench, covering abstract syntax ①, constraints ②, concrete syntax ③ and semantics of modeling languages (defined with generator as ④). The second way is graphical metamodeling, where a normal MetaEdit+ model automatically produces and processes the input for the third way, an XML import/export format for metamodels. The graphical way covers the abstract syntax and constraints of the language; the other ways cover all parts. The right side of Fig. 2 shows the resulting modeling tool: ⑤ a model created in a modeling editor of MetaEdit+ and ⑥ a sample of the code generated from this model.

6.2 EMF/Sirius

Sirius [16] is an Eclipse project that enables the development of graphical modeling environments by leveraging well-established technologies. Sirius is the natural evolution of GMF[6] and, despite the existence of other frameworks, including Eugenia,[7] and Graphiti,[8] it can be considered the most advanced meta-editor for the EMF ecosystem. Starting from a metamodel defined in Ecore (called domain model in Sirius), it allows a model-based specification of visual concrete syntax organized in viewpoints, i.e., models can be authored by means of different notations that suit the needs of various stakeholders. Sirius allows the conceptual separation between the abstract syntax and the corresponding representation(s) by means of Viewpoint Specification Models (VSMs), also referred to as mapping models. A mapping model consistently specifies the structure, appearance, and behavior of an editor according to the domain model and is stored in the `.odesign` format.

Because the EMF ecosystem is essentially a component-based, community-driven open-source endeavor, it consists of a large number of frameworks, tools, and languages that are typically developed as individual projects with little coordination among them. Thus, the ecosystem lacks a common infrastructure for natively managing artifact dependencies and traceability, making co-evolution management somewhat troublesome. Over the years, a corpus of research has been proposed for the co-evolution of metamodels and models (e.g., [3, 20, 23, 25–27, 60]), transformations (e.g., [30, 32, 61]), constraints (e.g., [62]), editors (e.g., [12, 13]), and syntaxes (e.g., [63]) in the EMF ecosystem. What emerges from these studies is that (i) each approach focuses on a specific category of artifact, requiring the modelers to become familiar with very diverse techniques, and (ii) such techniques cannot rely on integrated management of the dependencies, as found in MetaEdit+ and Jjodel. Therefore, while addressing the scenarios presented in the previous section, most of the artifact migrations have been done manually.

Figure 3 illustrates the components implementing the example outlined in Sect. 5. The metamodel is edited using the default tree-based editor ①, alongside a model instance ②. The odesign models ③ are instrumental in defining the graphical notation and the workbench tools. These tools facilitate the representation of state machine models ④.

Regarding generators, they are implemented using template-based languages ⑤. This category includes Acceleo[9], EGL [64], and Xpand [65]. Each of these languages is mature and has proven to be effective in practice. Acceleo, in particular, stands out as one of the most widely adopted tools, making it an exemplary representative for this category of generators in our scenarios. For simplicity, our discussion will primarily refer to the approach as EMF/Sirius, though it implicitly includes Acceleo. The results of applying the developed Acceleo template to the model are shown in ⑥.

6.3 Jjodel

Jjodel [17] is an Ecore compatible cloud-based, reflective modeling tool that tries to minimize accidental complexity and improve usability. Jjodel does not require expensive installation or maintenance, making modeling frameworks more approachable for new users and trainees. Jjodel is reflective and can reason about its structure and behavior, reducing costly operations like generation, compilation, and deployment of artifacts. It comes in an integrated environment with project management, where metamodels can be developed as a collection of viewpoints defining the abstract syntax, and multiple concrete syntaxes with associated editors. To reduce the cognitive load for the users, default syntaxes (and workbenches) for both metamodels and models are provided as topological notations, similar to UML class and object diagrams respectively.

Jjodel does not show the technical maturity of MetaEdit+ and EMF/Sirius as it is an academic effort. A major difference from the other two tools is its lack of support for metamodel invariants or constraints; Jjodel does however support multiplicity constraints on all kinds of relationships between metamodel elements. As to the generators, they are supported as a special kind of viewpoint via the templating mechanism supported by the tool.

[6] https://eclipse.dev/modeling/gmf/.

[7] https://eclipse.dev/epsilon/doc/eugenia/.

[8] https://eclipse.dev/graphiti/.

[9] https://eclipse.org/acceleo/.

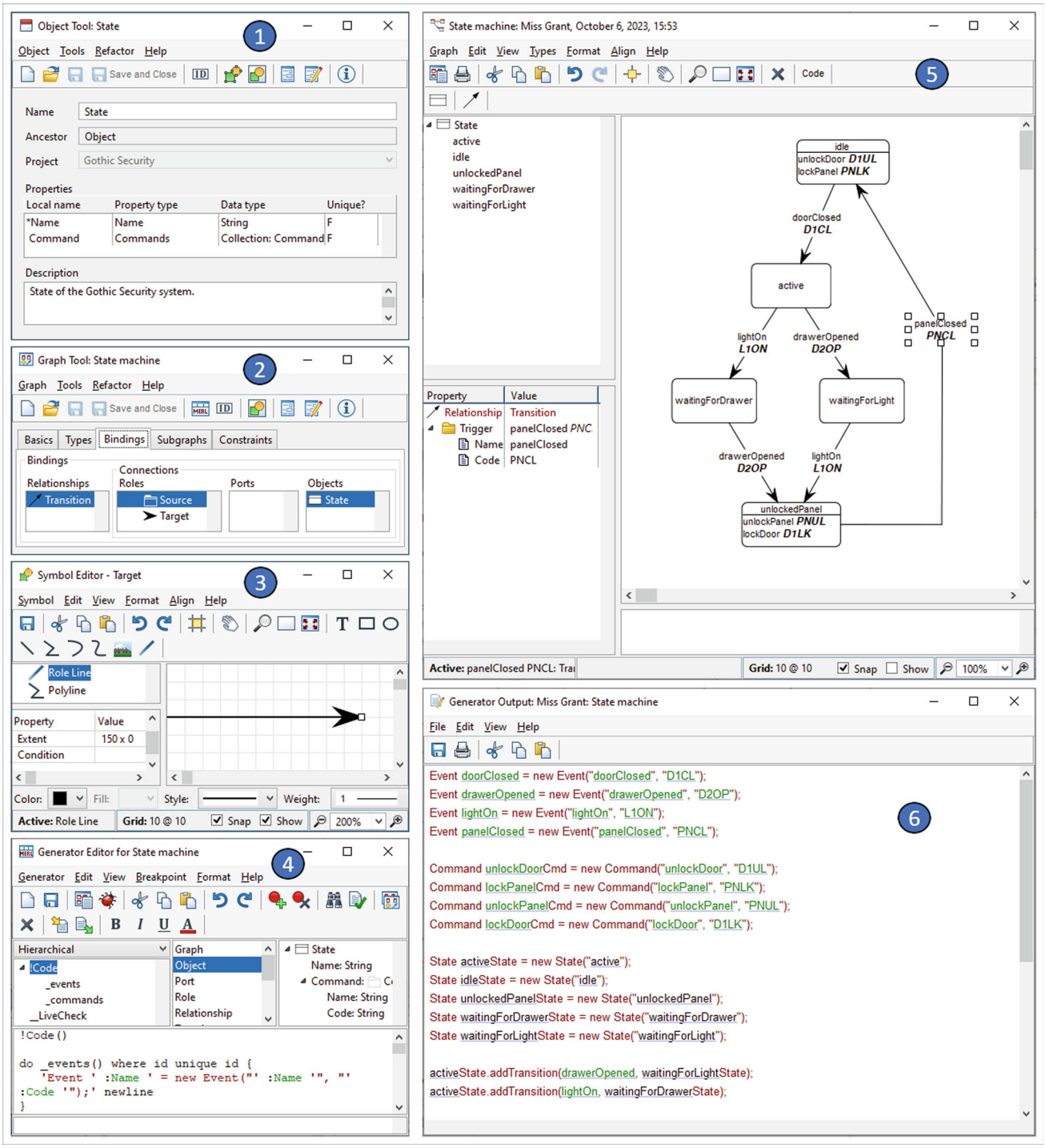


Fig. 2 The State Machine metamodel edited in MetaEdit+ (color figure online)

Figure 4 depicts an instance of the various components involved in developing the example outlined in Sect. 5 with Jjodel. Reading from the bottom, the topological metamodel editor ① edits the metamodel. The model instances can be viewed using the default tree-based editor ②, and once the graphical notation is defined ③ also in the resulting graphical editor ④. Template-based generators (similar to ③) can be made to produce code and textual reports ⑤.

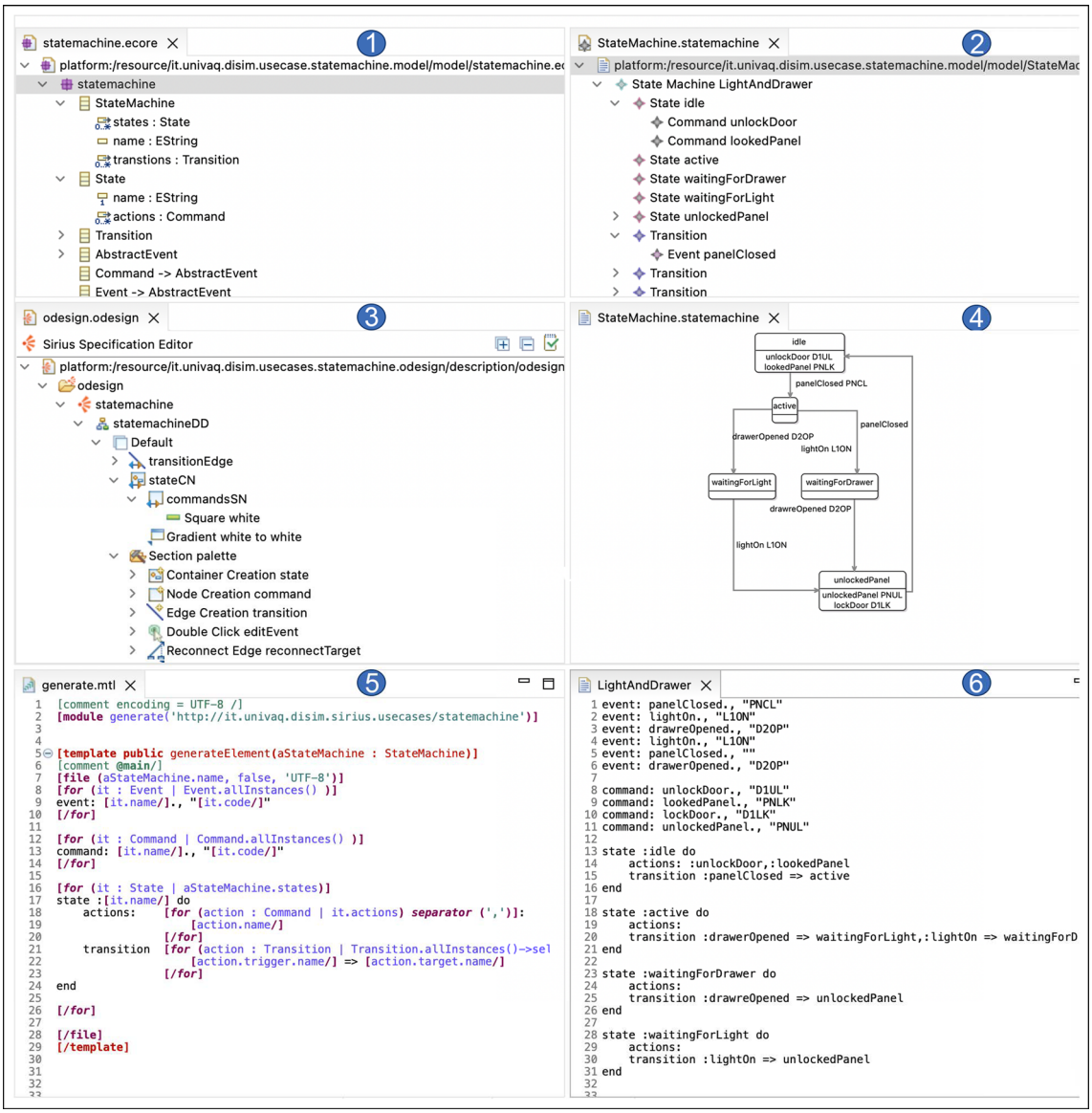


Fig. 3 The State Machine metamodel edited in EMF/Sirius (color figure online)

7 Evaluation execution

The evaluations were performed by users already experienced on each tool, using the normal tool functions available for all users. The co-evolution scenarios were performed by three people, one per tool, and results checked by another person. The results of each scenario and tool were presented and discussed among all authors and scores were given based on mutual decision. For the sake of reproducibility and ease of reference, the detailed results of the evaluation have been made available online, with language definitions and models versioned before and after each co-evolution scenario: see Sect. 8.1.4.

In the rest of this section, we go through the evolution scenarios presented in Sect. 4 in detail, discussing the co-evolution impact of the language changes. We will take the four scenarios of each 'Nature of change' in turn: adding, renaming, removing then changing, and within each will look at each tool, performing the three scenarios with that tool.

7.1 Adding new language elements: scenarios 1–3

Adding a new optional element to a language is typically easy from a model co-evolution point of view, as existing models remain valid after the change. However, depending on the tool there may be an impact on the other kinds of artifacts as described below.

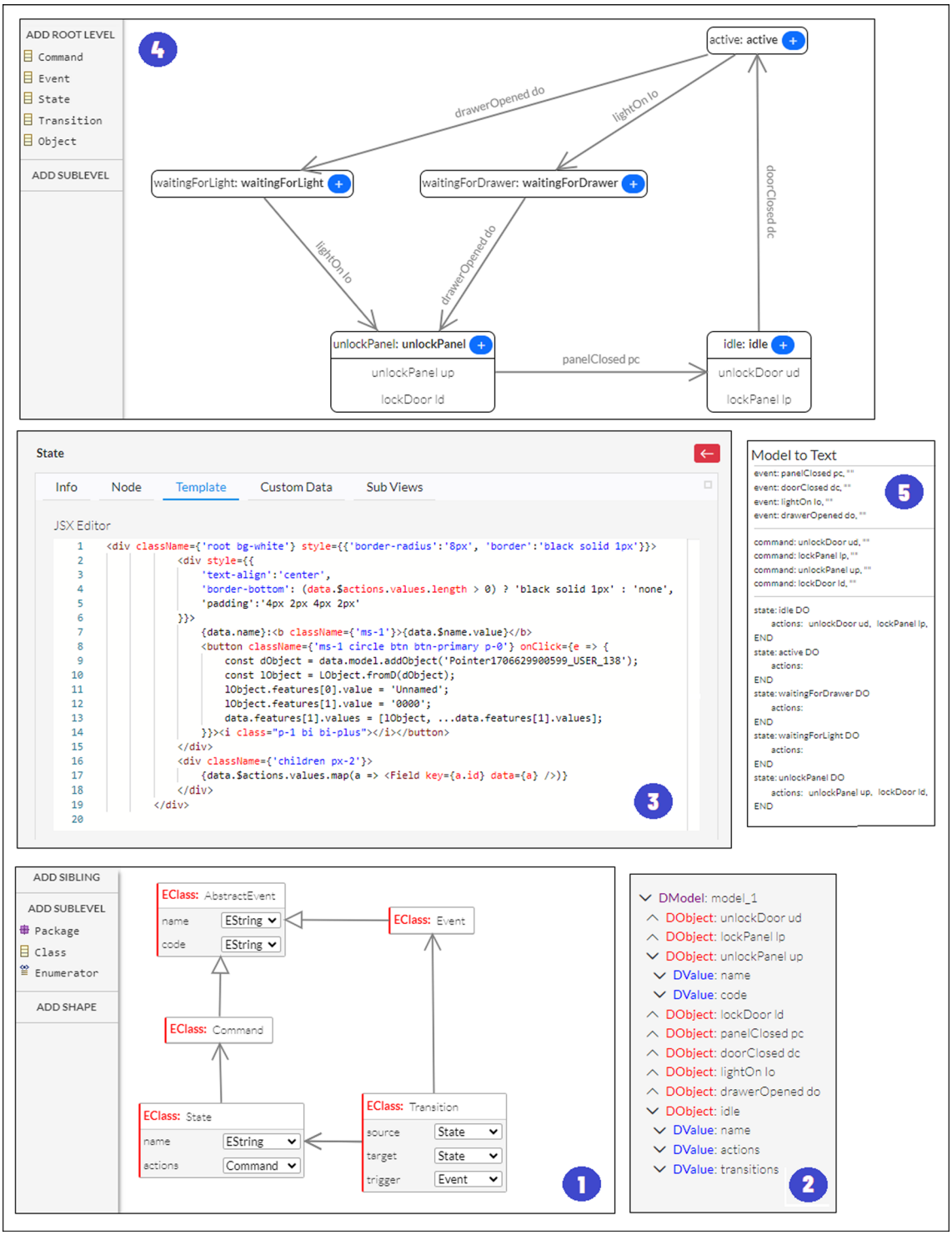


Fig. 4 The State Machine metamodel edited in Jjodel (color figure online)

MetaEdit+

To add a new metamodel element (scenario #1), the Graph Tool is used to add a new object type 'Reset' with a new property type containing a collection of Events ('Event' already exists in the metamodel). MetaEdit+ provides the editing functionality automatically, along with a simple default notation that the language engineer may change as desired.

Constraints set well-formedness rules to the language. In MetaEdit+, constraints include 1) bindings that say a relationship type can connect certain types of objects in certain types of roles, possibly via certain types of ports on the object, and 2) constraints on object occurrence, connectivity, ports and property uniqueness.

New constraints (#2) are added in the Graph Tool: an occurrence constraint that allows only one 'Reset' in a graph, and a connectivity constraint that allows a Reset to only be in one Transition. When a constraint is added, its influence on the existing models may need to be checked, as there may be models that do not satisfy the new constraint, e.g., by already having multiple Resets. In MetaEdit+ both models and metamodels are stored in the same repository, allowing language engineers to view and inspect the impact of their changes on models, before committing the changes and making them available for language users. This helps the language engineer to experiment, see the results of changes, and think what might be best from a modeler's point of view.

The new constraints restrict the set of valid models, so models with two or more Resets or Transitions from Resets are now invalid. There is no way to automatically make the models correct: deleting the extra Resets or Transitions would lose information. To assist modelers in updating after language evolution, all models calling for a modeler's decision can be listed or annotated. In MetaEdit+ this can be accomplished by adding a conditional graphical annotation in the Symbol Editor, or by a generator listing model elements that do not meet the constraints. An example of this is shown at the bottom of the editor in Fig. 5a. In this case, MetaEdit+ would report on models having Resets that do not meet the constraints. This fulfills the most highly demanded tool feature in [42]: to highlight invalid model elements and show associated error messages.

To add the new notation (#3), the symbol for 'Reset' is created in the Symbol Editor by drawing it as vector graphics; it could also be imported from an SVG or bitmap file.

After adding these three language elements, all editors have full functionality, and all existing models open and update automatically. In the case of the new constraint, modelers are guided to update model elements that violate the constraint.

EMF/Sirius

To add a metaclass (#1) in Ecore, the tree-based editor is used to instantiate the corresponding EClass. To allow the new metaclass to be instantiated, Ecore typically requires that it is part of a containment, i.e., Reset is contained by the root metaclass StateMachine. No adaptation is required on the existing constraints and notation, i.e., the Reset element is not yet represented at this stage and Sirius does not feature any default syntax. Generators are still valid; however they will not yet generate anything specifically for Reset instances. As for the default syntax, Sirius does not create a default entry in the tooling palette for the created element. No changes are needed for models because the Reset metaclass is optional, i.e., the lower bound of its containment cardinality is zero.

Because the constraints to be added related to Reset correspond to multiplicity constraints in Ecore, adding a new constraint (#2) means to change the cardinality of the Reset containment from *0-to-many* to *0-to-1*. Such a modification does not have any impact on the metamodel, constraints, and notation. However, the Acceleo template for generation must be updated because the existing iteration over the Reset instances throws an exception. The tooling palette still does not feature Reset, although it keeps working for the earlier metaclasses.

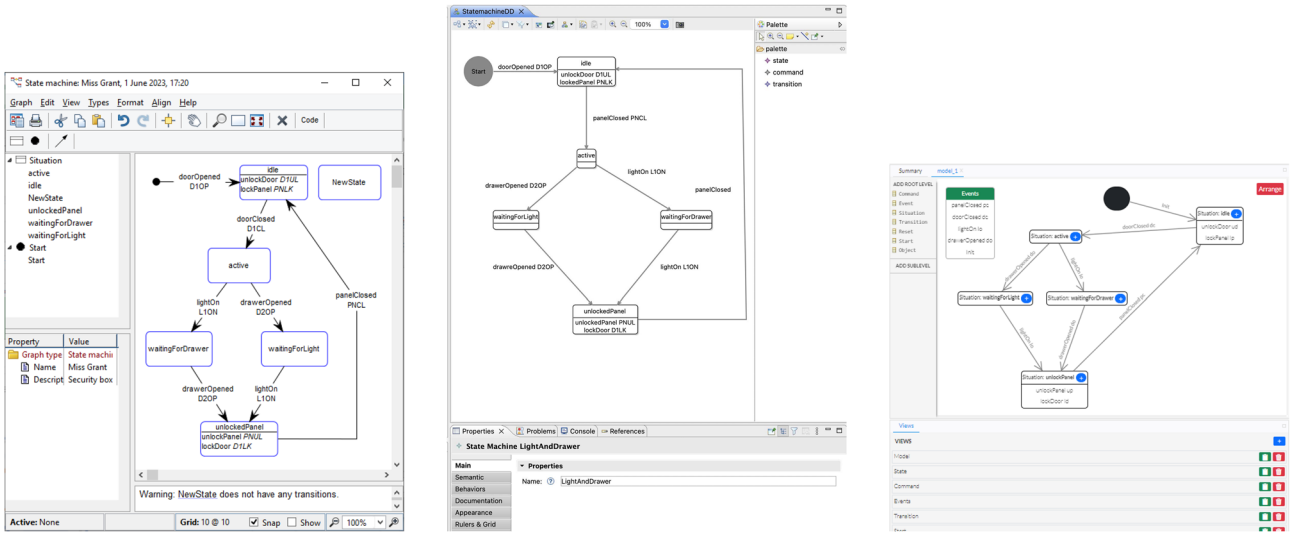
The language change may affect the conformance of models depending on the number of the existing Reset instances, i.e., if more than one instance exists, then the model does not conform to the new version of the metamodel and adaptation steps are needed. When multiple Reset instances are present, the model should undergo adaptation. The modeler has to choose the Reset instance to be preserved in the evolved model while discarding the remaining ones. If the operation is automated, some evolution heuristic is typically defined to identify the Reset instance to be kept in the model, e.g., the first added Reset instance.

A notation for Reset can next be added (#3) by modifying the odesign specification that includes the graphical representation and the corresponding palette entry. All other definitions remain unaffected.

Jjodel

The Jjodel metamodel notation is essentially based on Ecore. However, it does not require that metaclasses are contained in a root element. Such a slight difference and its inherently reflective nature make Jjodel more similar to MetaEdit+ than to EMF/Sirius.

When adding a new metaclass Reset (#1), Jjodel's default graphical syntax and editor will be used. Because no contain-



(a) MetaEdit+ (b) EMF/Sirius (c) Jjodel

Fig. 5 Tools after 12 co-evolution scenarios (color figure online)

ment is required, the rest of the metamodel remains entirely unaffected by the change. Jjodel does not yet support constraints, so there is no impact there either. Existing notations are also unaffected. The generator keeps working, although Reset instances are ignored. The tooling is unaffected as the Jjodel palette and creation tools are automatically configured; and, finally, models are still valid because Reset is an optional element.

Scenario #2 cannot be performed because of Jjodel's lack of support for constraints. To allow the subsequent scenarios, Reset is extended with a reference to State called transition. This has no repercussions on the rest of the metamodel. The notation for the StateMachine must however be extended with code to display Reset Transitions, as they are stored differently from other Transitions. Similarly, the generator must be modified to cover Transitions contained in Resets. As for the previous scenario, the tooling and the models are unaffected.

Adding the notation for Reset (#3) is accomplished by adding a View for Reset and specifying the desired graphical and textual elements in a short Template containing the React JSX[10] combination of HTML and JavaScript. This has no impact on other areas of the language or models.

7.2 Renaming language elements: scenarios 4–6

In language workbenches in general, renaming an element in the metamodel may influence concrete syntax, constraints on the element, and often how semantics is defined. It may also influence existing models, if their metamodel references are based on names rather than more opaque IDs.

MetaEdit+

In MetaEdit+ renaming 'State' to 'Situation' (#4) in the metamodeling tool automatically updates the definitions of related constraints. If there are related generators, they need to be updated with find and replace. If there are several languages with different 'State' concepts the search can be limited to a given language. Updating generators is not needed if the generator is not bound explicitly to the name of the metamodel element. After renaming an element in the metamodel, the constraints, notation, tools and models update automatically.

Renaming a constraint (#5) does not occur in MetaEdit+, as constraints do not have names.

Renaming notation (#6) is also not normally encountered, as symbols used for the notation in MetaEdit+ are directly related to language elements and do not have names. For more complex cases, a symbol can however also be stored by name in a library, and another symbol can incorporate it from the symbol library by referencing it by name in a template. By renaming a 'Rectangle' symbol to 'BlackRectangle' in the place where it is referenced, both the rename and reference update are accomplished in one operation. The new notation is automatically reflected to models and modeling tools, with no adverse effects elsewhere.

To summarize: after the renaming scenarios, all tools of MetaEdit+ have full functionality and models are fully updated automatically. In scenario #4, generators required simple updates.

[10] https://www.w3schools.com/react/react_jsx.asp.

EMF/Sirius

In contrast with MetaEdit+, any renaming (#4, #5, #6) in Ecore has repercussions throughout the ecosystems. In particular, the notation defined in the odesign must be adapted otherwise the editor does not visualize the old State instances as Situation ones. Moreover, also the generator navigational expressions must be renamed accordingly as well as the tooling and the models. In scenario 4, involving the renaming of State to Situation, the validity of the existing models must be restored by adapting them. Only models that have been consistently modified can be accessed and edited with both the default EMF tree-based editor and the adapted Sirius workbench. Specifically, models that fail to rename existing State instances to Situation are incompatible with the modified metamodel and the revised Sirius workbench and, consequently, cannot be opened. In the supporting repository, scenarios 4–6 have been committed at once due to the simplicity of the modifications, e.g., no constraint and notation renamings occurred.

Jjodel

The metaclass renaming (#4) impacts the ecosystem differently depending on the identification method used. In Jjodel, notation and generators can reference elements in one of two ways: either by IDs such as 'Pointer1706784131828_USER_140' and 'Pointer1706784131827_USER_138' or names such as 'State' and 'Situation.' While names make views and generators more readable and intuitive, using IDs can insulate the system from the potential disruptions that renaming operations might cause. For clarity, we have used names in the notation and references in generators, respectively, leading to different scores as summarized in Table 5. Jjodel does not support the other renamings (#5, #6) as it does not support named constraints, and renaming syntax view in notation does not impact the ecosystem.

7.3 Removing language elements: scenarios 7–9

Removing an element from the metamodel (#7), like 'Reset', typically has a significant impact on other parts of the language and models. Language engineers can first consider if it is better just to hide the metamodel element or make it no longer instantiable, rather than delete it and all its instances permanently. The gentler approach allows existing model data to be used for example when generating code—after all, the generator support for them already exists and works. One bonus here is that if it is later found that removal was not a good idea it is possible to bring the removed parts back—and with good tool support this will also fully restore their instances.

This approach of deprecating rather than hard deletion is popular with language users, allowing them to see and update design data created earlier, while guiding them not to use the old language concept anymore. The downside of this approach is that models will not fully conform to the current metamodel: anathema to more theoretical worldviews and requiring extra work in tools to be able to cope.

MetaEdit+

Removing Reset from the metamodel (#7) is done in the Graph Tool by removing 'Reset' from the list of State Machine language elements. Reset itself remains defined. On the model level, instances are still visible and the language engineer or modeler can remove them from models, or they can be removed by using model transformation with the MetaEdit+ API [56]. If the removal involves decisions dependent on the model context, the language engineer can implement model check functionality similarly to that made earlier when adding new constraints (#2). In this case, the existing warning for two or more Resets was changed to warn for any Resets. Removal rarely calls for changes to code generators, and none was needed here.

Removing a constraint or binding (#8), e.g., that 'Reset' is allowed to be connected to 'Situation', is done in MetaEdit+ by removing it from the list of bindings in Graph Tool. Removing a constraint generally broadens the set of possible models, and so does not require additional actions from the language engineer nor from language users, but removing a binding narrows the set of possible models, so it may be useful to provide deprecation guidance as in scenarios #2 and #7.

Removing an element from the metamodel normally removes its notation automatically too. If only the notation is removed, as with Reset's symbol in scenario #9, the default notation will be used in its place.

Removing language elements in scenarios #7–9 calls for normal language engineering tasks. Since deprecation guidance is provided for scenarios #7 and #8, and models, tools and generators continue to work, language users do not necessarily need to take any actions in these cases.

EMF/Sirius

Removing a metaclass in Ecore (#7) requires generators, tooling, and models to be manually updated to restore their consistency. In more detail, once the Reset metaclass is deleted, the StateMachine containment must be removed, and the generator template fragments containing navigational expressions referring to the Reset metaclass must be deleted. After removing the Reset metaclass in scenario #7, removing a cardinality constraint (#8) in Ecore does not occur as the constraints are automatically removed in the previous step.

Removing the notation corresponding to the Reset metaclass (#9) has a similar impact as the previous two scenarios. In addition, the tooling palette should be adapted as the Reset metaclass can no longer be instantiated.

While the removals in scenarios 7–9 have an impact on many different artifacts, as reported in Table 4, the consequent adaptations are rather straightforward as they are relatively small and do not demand any specialized knowledge.

Similarly to scenarios 4–6, scenarios 7–9 have been committed simultaneously since, for each scenario, only simple atomic changes are performed.

Jjodel

Removing Reset from the metamodel (#7) does not automatically delete its instances; they become untyped ("*shapeless*") and remain available for later use, including explicit deletion, somewhat similarly to MetaEdit+. Constraints are unaffected; notation definitions are unaffected, but although the notation definition for Reset remains present it is not applied to any previous instances of Reset, as they are now untyped and thus not seen as Reset instances.

Similarly to previous cases involving constraints, scenario #8 cannot be executed because of the lack of support in Jjodel.

Removing the graphical representation for Reset in the notation (#9) in Jjodel corresponds to disassociating it from the notation viewpoint, i.e., the representation is still present but no longer associated with the current notation. Such a modification does not affect the metamodel and the other components.

7.4 Changing reference on existing language elements: scenarios 10–12

The last three scenarios are the most complex modifications covered in this paper, because they are composite changes that, depending on the tools, may involve moving concepts, adding subclasses, and more complex graphical notations.

MetaEdit+

Changing a reference to an existing element in the metamodel, like in #10 moving the 'Trigger' Event property from the 'Transition' relationship to the 'Source' role, is more challenging than a simple removal and addition. In this case the model co-evolution could in theory be automated, as each Transition has exactly one Source role (see Sect. 7.6.1 for details). Deprecation can still be used to good effect: we can allow the Trigger Event property to remain in the Transition, as well as adding it to Source. In that case, it seems most sensible to make the new property (when provided) override the old, and to flag as errors or at least warnings cases where both are provided—at least if they specify different Events.

After this change the 'Transition' relationship still also has the 'Trigger' information, and generators use that information. Keeping 'Trigger' in 'Transition' is useful for the transition phase, so that current Trigger information can be moved to 'Source' role. This can be done manually by cutting and pasting the existing Trigger Event from the Transition to the Source role, or by using model transformation with the MetaEdit+ API [56]. Language engineers can also prevent creating new 'Triggers' in 'Transitions' by making the property type read-only there. They can also give deprecation guidance with an annotation or report as in #2.

Changing an existing constraint calls for changing a reference to an existing element. To conduct scenario #11, the language engineer must first add a new object type ('Start'), in a similar way to scenario #1. Next in the Graph Tool the 'Start' is added to the existing binding constraint by including it in the 'Source' role alongside 'Situation.' To finalize the scenario, the existing constraints set in step #2 for 'Reset' are updated by changing them to 'Start.'

Finally, changing a notation reference (#12) means choosing another symbol for the notation or its parts. In MetaEdit+ the template subsymbol can be replaced by opening the Symbol Editor for 'Situation', opening the template element and changing it to use another subsymbol from the library. (The symbol deliberately uses a template so we can demonstrate the more complex case.)

During the evolution through these changes, editors continue to work without errors or omissions, and old models open normally. For scenario #10, modelers correctly can no longer add Trigger information to Transitions, and they see notifications to update existing Triggers. If model transformation is used for #10 existing models can also be updated automatically, moving Trigger information to the Source role.

EMF/Sirius

In Ecore, moving the Transition's Trigger to the Source end of the relationship (#10) is not directly possible, as there is no explicit concept for the Source end of the relationship. The closest corresponding action found was to move the Transition from being contained by the StateMachine to being contained by its source Situation. This requires several modifications affecting the metamodel structure. Because the cardinalities remain unchanged, the constraints are not affected. Also, the notation, the generator, and the tooling must be adapted as broken references must be fixed. Finally, the models do not conform to the new version of the metamodel and, therefore, must be migrated.

The constraint change (#11) does not occur in Ecore. However, we added a new metaclass Start, as a specialization of

Situation, to achieve the same result. This impacts all components except the models, where no change is necessarily required because Start is an optional element. This is because we limited the constraints' expressiveness to cardinality and typing requirements, leaving more expressive notations like OCL out of the experiment. In this respect, Start inheriting from Situation can be regarded as a typing requirement.

The adjustment of the notation (#12) must take into account all the previous changes (#10 and #11), namely the relocation of Transition within Situation and the establishment of a subtype. Consequently, it affects the notation and tooling by necessitating the inclusion of a graphical representation for the Start element, the ability of the tooling to create instances of Start, and the editor's awareness that the Transition is now located within the Situation rather than the StateMachine.

Jjodel

In Jjodel, relocating the Trigger of the Transition to the Source end of the relationship (#10) is not directly possible. This is similar to EMF/Sirius, so the same replacement task is used: transferring the containment of the Transition to the Situation. The change does not impact on the rest of the ecosystem.

To perform scenario #11 we add a new metaclass Start extending Situation. This is analogous to the Start-Situation case in the previous section, i.e., the necessary typing requirement is obtained by specializing State with Start. In addition, the shapeless objects originating from the deletion of Reset (#7) are given the Start typing. The metamodel is affected by the modification, whereas the notation remains unchanged. The generator must be adapted to support Start in place of Reset. The tooling remains unchanged, and the models do not need to be adapted because Start is an optional element.

To customize the notation for Start (#12), we reused the Reset representation disabled in #9 and provided it the requested visual representation. As usual with the notation in Jjodel, such a modification leaves the rest of the components unaffected.

7.5 Tools' co-evolution scores

Figure 5 illustrates the co-evolved language and model in tools after the 12 scenarios. In the following subsections, Tables 3, 4 and 5 summarize the results of the evaluation for each tool, respectively. As depicted in Fig. 6, the overall score for each scenario is given first in bold, followed by the subscores for individual impact locations. All scores are in the range of 1–5, with 5 being best. The first three subscores state the impact of change on any part of the metamodel, constraints and notation respectively. The next three subscores cover the impact on generators, tooling and models. With industrial-scale modeling in mind, it is worth noting that losing points in the last subscore possibly requires work by many modelers on many models, whereas the five earlier subscores only require work by the language engineer(s) on the language artifacts.

Table 3 MetaEdit+ co-evolution evaluation scores: overall in bold, then subscores for metamodel, constraints, notation—generator, tool, model (color figure online)

Location of change ↓	*Nature of change*			
	Add	**Rename**	**Remove**	**Change**
Metamodel	**5** 555—555	**2** 555—255	**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

Table 4 Sirius co-evolution evaluation scores: overall in bold, then subscores for metamodel, constraints, notation—generator, tool, model (color figure online)

Location of change ↓	*Nature of change*			
	Add	**Rename**	**Remove**	**Change**
Metamodel	**2** 555—525	**1** 552—155	**1** 555—131	**1** 552—121
Constraints	**1** 555—221	—	**1** 555—131	**2** 352—325
Notation	**5** 555—555	—	**5** 555—555	**5** 555—555

Table 5 Jjodel co-evolution evaluation scores: overall in bold, then subscores for metamodel, constraints, notation—generator, tool, model (color figure online)

Location of change ↓	*Nature of change*			
	Add	**Rename**	**Remove**	**Change**
Metamodel	**5** 555—555	**2** 552—555	**3** 555—3½33½	**5** 555—555
Constraints	**X** xxx—xxx	**X** xxx—xxx	**X** xxx—xxx	**2** 355—255
Notation	**5** 555—555	—	**5** 555—555	**5** 555—555

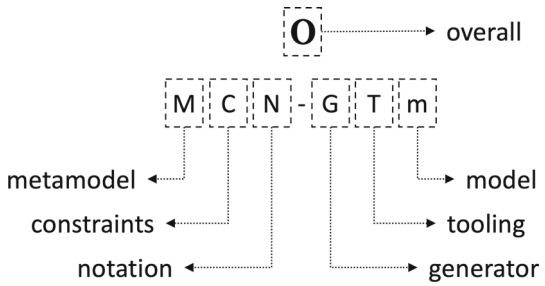


Fig. 6 Key to the scores in a scenario cell: subscores by location of impact, minimum as overall score (color figure online)

The table cells are colored light green for fully automated (5), lightish green for automation as full as seems desirable (4½), and mid-green for cases in which the only adverse impact is in generator or model co-evolution, where human interaction is needed (4). A mid yellow is used for (3½), a deeper yellow for (3), orange for (2) and red for (1). In grayscale the colors run from (5) lightest to (1) darkest.

We also found three different kinds of situation where a tool was unable to complete a scenario. In the first, a similar task elsewhere in the language could be found, and that was scored as normal. In the second, only applicable for Rename scenarios, the elements for that location of change in that tool did not have names, so no problems from renaming were possible: for these we marked a dash '–' on a light gray background. In the third, the tool did not support the specified functionality, and no clearly parallel similar functionality: for these we marked an 'X' on a dark gray background.

7.5.1 MetaEdit+ scores

In none of the cases does the functionality of editors or other tools in MetaEdit+ break or show incorrect or non-working UI elements. In the case of multiple people using the language the result would be the same: they all automatically get the updated language version and same co-evolution success.

In no scenario is there an adverse impact on the metamodel, constraints or notation, nor on the tool functionality, so the co-evolution score for these is always the highest, 5. In most cases the models too update automatically when the language is changed. In no cases are the models damaged, but in three cases the change is such that a fully automatic model update would not be desirable, and in one case not even possible, so deprecation advice is provided for models using the old style, and both new and old style can coexist and generate code correctly. Since existing models and generated code remain valid and deprecation guidance is provided, these cases have a co-evolution score of 4½ in Table 3. In scenario 4, the renaming of an element in the metamodel requires a manual find and replace to update the generators, giving a co-evolution score of 2.

Where deprecation with manual update advice was provided, an alternative would be to automate model transformations with the MetaEdit+ API. The API was not used in the co-evolution cases described here, but if used it would change the score to 5 in scenarios 7, 8, and 10, where automation without additional modeler input may be acceptable (see Sect. 7.6.1 below for details).

7.5.2 EMF/Sirius scores

In a previous work [13], we have already explored the resilience capabilities of Sirius. This paper confirms that certain limitations make complex evolution patterns challenging, as summarized in Table 4. In particular, in no scenarios are the constraints adversely affected, whereas a moderate adverse impact on the metamodel is present only in scenario 11, where we add the Start metaclass as a specialization of Situation, which contains a list of (references to) Action(s). It is worth noting that by constraint we mean usual requirements about relation cardinality and typing, which means we are not considering any predicate expressed, e.g., in OCL. The metamodel and the constraints represent the least impacted artifacts in our co-evolution scenario-based analysis.

The notation is negatively affected in scenarios 4, 10, and 11 as the changes performed require adaptations to restore the correct functionalities of the environment. Please note that in certain scenarios the adaptation needed is required in a subsequent scenario, e.g., Reset is added in scenario 1 and its notation is added in scenario 3. In such cases, adapting the notation as a response to a change is not reported as an adverse impact.

Because maintaining consistency among the EMF components is the responsibility of the language engineer, the Acceleo-based generator and the tooling have been adversely impacted in most of the scenarios. There is no automated mechanism that keeps things aligned: only in scenarios 1, 3, 9, and 12 do they remain unaffected. In addition, the tooling remained unaffected by scenario 9. In particular, the generator keeps working but fully ignores the meta-elements added in the various scenarios, e.g., in scenario 1 Reset is added in the metamodel and the generator is still working but completely ignores its instances.

Finally, the model is the artifact that requires considerable adaptations in scenarios 2, 7, 8, and 10 because of the most adverse impact due to the operated refactorings in which they scored only 1. Such an impact is explained by the fact that models and metamodels are managed by almost completely independent editors, and these do not exploit the dependencies due to the conformance relation, used to good effect in the more integrated environments of MetaEdit+ and Jjodel.

7.5.3 Jjodel scores

Similarly to MetaEdit+, Jjodel is an integrated environment with better management of the artifact dependencies to leverage some automated tasks. This enhances its capability to handle evolving scenarios. In particular, only in scenario 11 does it drop points on the metamodel subscore: there the metamodel must be rearranged since it is not enough to add the Start metaclass, it also has to be a specialization of Situation. Constraints are not supported and are thus also not affected in any of the scenarios. The notation is affected only in scenario 4, where the JavaScript code is not automatically adapted when the metaclass name is changed.

Scenarios 2, 7, and 11 have an adverse impact on the generator: in scenarios 2 and 11, the generator does not provide all the functionalities; in scenario 7, we remove the Reset metaclass, and the generator ignores its instances. However, such instances are denoted as *untyped*, and the generator fails to notify the existence of such instances and the fact that the rule for generating code out of the Reset instances is still present, although not used. The tooling is almost unaffected by the changes, losing points only in scenario 7 as it does not provide support to already existing artifacts. Finally, the model must be adapted in scenarios 2 and 11.

7.6 Effort to perform co-evolution

We evaluated the effort of co-evolution by measuring the time needed to conduct each scenario. They are summarized in Table 6 and detailed for each tool in the following sections.

7.6.1 Effort with MetaEdit+

Completing the evaluation framework's scenarios was straightforward in MetaEdit+, both for implementing the language changes and assessing their impact. The evaluation was not time-consuming, taking 32 minutes to implement the 12 scenarios. The times were very similar over three separate repetitions. Each scenario was completed before the next, including any necessary manual updates to models or generators, and writing a version comment. This time thus includes the language engineer's work to add co-evolution guidance and update any generators impacted, and the language user's work to update the model according to the co-evolution guidance. In our case the same person performed both roles of language engineer and language user.

The detailed figures of the effort were taken from the version timestamps of GitHub and are summarized in Table 6. Individual scenarios took just some minutes and the steps that took the most time also included the most modeling work, like adding several Reset events in scenario 1 or manually moving Triggers from six transition relationships to equivalent source roles in scenario 10.

We deliberately chose to follow a deprecation approach, and to provide the best possible deprecation guidance to modelers—even though this increased the times significantly. This choice is based on our experience with industrial use, where forcibly losing information or requiring everybody's models to be updated at the same time are often not possible or desired. The effort to update models manually naturally increases when models are larger. However, if the deprecated structures are allowed to remain, the cost of updating is removed. Where that is not the case, automating the updates can be preferred at the point where the size of models makes the overall cost of automation smaller than the overall cost of manual updates.

For the scenarios in which updates were performed manually, it is also possible to use model transformations in scenarios 7, 8 and 10. For such purposes MetaEdit+ provides an API [56] with operations that can handle model changes on both conceptual data and representations. The API provides co-evolution operations that can be called from virtually any language. The time taken to write a transformation can easily be recouped when running it on models of any size. For example, manual model updates for scenario 10 took around a minute on a small model, but with the API the necessary changes can be specified in under 10 lines. Running the API code on the 3456 transitions of the larger example model set completes the updates in 17 sec[11]. The effort to make the same changes manually on that larger model set would vary depending on the person and experience, but for scenario 10 we estimate around 6½ hours. As mentioned earlier, the use of the API would raise scenarios 7, 8 and 10's scores from 4½ to 5.

MetaEdit+ thus supports automatic updates (handled entirely by the tool), deprecation without model updates, deprecation with manual updates on a timetable decided by the organization, and automated updates (automation created by the language designer). Even with the largest models, performance in all metamodeling and modeling tasks remains excellent.

7.6.2 Effort with EMF/Sirius

While EMF/Sirius is very efficient in producing a prototypical environment with little effort, maintaining and consolidating such tools can be troublesome as evolution stages must consider dependencies that, in EMF/Sirius, are not leveraged to a first-class status [13]. As a consequence, it does not come as a surprise that the twelve scenarios to be executed required 56 min as summarized in Table 6. This time somewhat compares to the effort to design the model editor with Sirius from scratch for the state machine metamodel.

The table also includes the time needed for the adaptation of models. As long as the models are relatively small, migrations can be obtained manually in a cost-effective way. However, when we consider much larger models (e.g., 2880 states, 3456 transitions and 12096 property values), manual adaptations are not always feasible and such procedures must be designed *ad hoc*, e.g., in terms of model transformations. Nevertheless, we evolved large models by strategically modifying XMI representations by finding and replacing terms. This approach was chosen because the changes required were straightforward enough to handle manually and efficiently,

[11] The model transformations were performed with a PC running Windows 10 with a 3.20 GHz i7-8700 CPU and 16 GB RAM.

Table 6 Time in minutes for each scenario, one small diagram as the model

Tool	*Location of change* ↓	*Nature of change* **Add**	**Rename**	**Remove**	**Change**
MetaEdit+	**Metamodel**	5	2	2	8
	Constraints	2		3	4
	Notation	2	1	2	1
EMF/Sirius	**Metamodel**	2	2	1	10
	Constraints	5		1	4
	Notation	15		1	15
Jjodel	**Metamodel**	1	1	1	1
	Constraints				1
	Notation	1		1	2

ensuring the evolution of each model without the need for co-evolving transformation tools or processes. As a rule, Sirius does not expose any scalability issue in storing and loading large models.

7.6.3 Effort with Jjodel

The Jjodel performance in executing the evolution scenarios compares to the performance of MetaEdit+. This is because Jjodel is an integrated environment with some native dependency management, in contrast with EMF/Sirius, which consists of several relatively sparse and uncoordinated components. As illustrated in Table 6, the overall time for executing the twelve scenarios is less than 10 min, which is considerably less than with EMF/Sirius and somewhat less than with MetaEdit+. This is due to (i) the efficient integration between the metamodel definition, the model editor, and the model being edited, (ii) to the usability of the concrete syntax notation, which is based on well-known front-end languages, including React JSX, and finally (iii) the fact that only eight scenarios were executed as three could not be managed by the current state of Jjodel, and one was unnecessary.

As to larger models, Jjodel fails in loading and storing models efficiently. In fact, models in Jjodel are serialized as edit scripts, i.e., sequences of atomic operations that manipulate models and execute the evolution scenarios. However, because of the size of the models, the amount of atomic operations to be executed requires considerable computational load.

Consequently, for large models that require complex transformations or extensive testing with mutants, the current version of Jjodel may not be the optimal tool. Instead, users of Jjodel must rely on its co-evolution support for smaller or less complex models, where manual updates remain manageable and the system's interactive tools can still provide value.

8 Discussion

The previous section described the execution of the evaluation per tool. In this section we bring the results together to discuss across tools, seeing what is common between them, where they differ, and what this motivates. We then return to the framework itself, discussing and reflecting on its application, and seeing what worked and where it might be improved.

8.1 Tool results discussion

With three tools, twelve scenarios for each tool, and six subscores and a time for each scenario for each tool, there is a total of 252 data points to present and compare. The times and the scores per tool are already presented in the tables in the previous section, with each scenario's six subscores also summarized by a single overall score. The tables allow us to see each tool's performance on its own, but are not ideal for comparing across tools. We tried several ways to combine the data, and found the most insight and visibility were given by radar charts. Other visualizations are available in the Supplementary Material [52].

Practice varies as to whether an ordinal variable, such as the subscores here, can be summed or averaged. A strictly theoretical approach would rule out anything other than minimum, maximum, mode or median. However, with many data points, the minimum and the maximum will be dominated by outliers, effectively throwing away most of the data. The mode or median also often fails to account for all values. In this case, taking the mode or median will, like the maximum, give a value of 5 for all scenarios across all tools. We found that the arithmetic mean gave the most useful visualization, and the uncertainty of the precise statistical nature of the scale was less of a problem in a radar chart, which presents more of an intuitive impression rather than a precise numeric score.

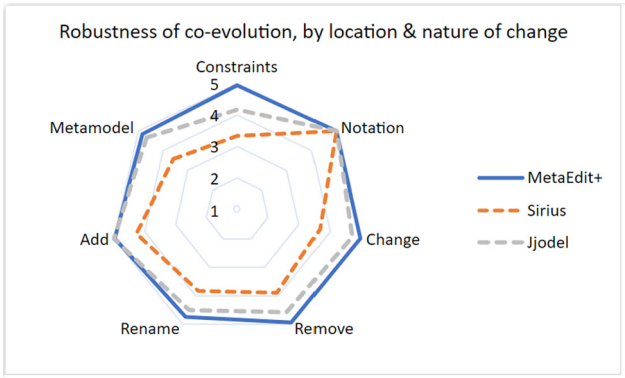


Fig. 7 Robustness of co-evolution, by location of change and nature of change (color figure online)

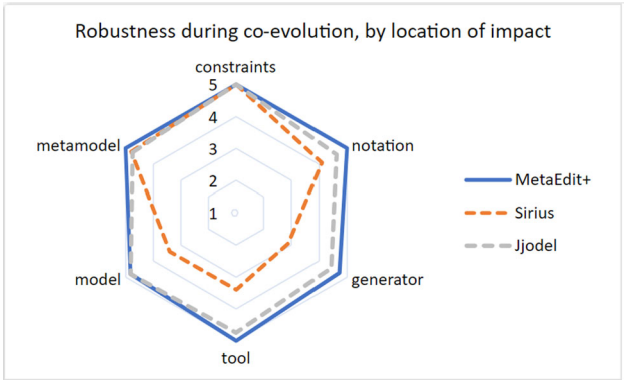


Fig. 8 Robustness during co-evolution, by location of impact (color figure online)

8.1.1 Location and nature of change

The first two aspects, location of change and nature of change, could usefully be displayed in the same radar chart. The top half shows the location of change: was the change made to the metamodel, constraints or notation. The bottom half shows the nature of change: did the scenario add, rename, remove or change an element. The three tools each have their own line, showing the pattern of their scores. The score is taken as the arithmetic mean of all the location of impact subscores for that location or nature of change. Each score in the top half thus combines 24 subscores, and in the bottom half 18 subscores.

As can be seen from Fig. 7, MetaEdit+ has the highest scores overall, with Jjodel close behind. Sirius is significantly further behind, but even its scores are all above the mid-point of the scale.

All of the tools have perfect scores on changes to the notation: none of the assessed locations of impact appears to be dependent on the notation. For the nature of change, all tools coped best with Add: adding new things to the language is indeed often the most common operation in practice, so good support there will be welcomed by users.

8.1.2 Location of impact

The third aspect, location of impact, has six subscores and was thus well-suited to having its own radar chart. The top half of the chart shows the locations most seen by the metamodeler, and the bottom half those most seen by the modeler. Once again, the score for each tool is taken as the arithmetic mean of the subscores, in this case that location of impact subscore from each of the 12 scenarios.

As can be seen from Fig. 8, MetaEdit+ again has the highest scores overall, with Jjodel slightly closer behind. Sirius is again significantly further behind, more so on the bottom half. All of the tools have perfect scores on the constraints, but in reality Jjodel was unable to complete three of the four scenarios, as it lacks most constraint functionality. The metamodel (abstract syntax) is rarely damaged by changes: while many other artifacts depend on the metamodel, the metamodel itself does not depend on other artifacts, and interdependencies within the metamodel seem to be handled relatively well.

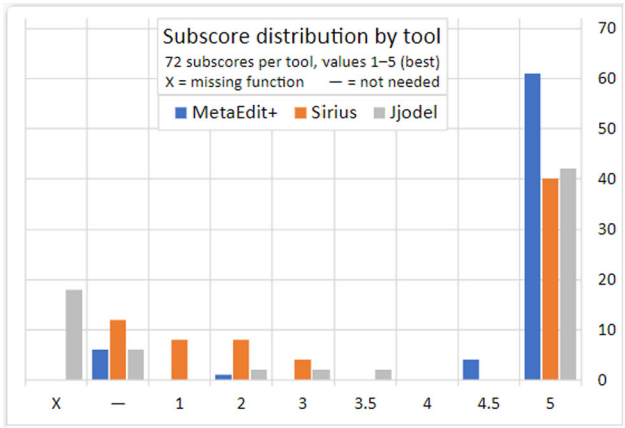


Fig. 9 Subscore distribution by tool (color figure online)

8.1.3 Scoring distribution

As we saw above, the radar chart cannot sensibly include scores for scenarios that a tool was unable to complete. A histogram can help to give a better overall comparison, showing how often each tool was able to obtain the maximum subscore of 5 (and other values). The histogram is also strictly legal for ordinal variables.

Figure 9 thus shows some information that is not visible in the radar charts above. MetaEdit+ obtains the maximum score of 5 about 50% more often than Sirius or Jjodel. Jjodel misses out where it is unable to accomplish the constraint task (the gray 'X' column), and Sirius where it can accomplish a task but leaves significant co-evolution problems (orange columns 1–3).

8.1.4 Reproducibility and replicability of the Evaluation

To facilitate the reproducibility and replicability of our study, and to provide a clear trace of the application of our co-evolution framework, we have organized our evaluation results into three distinct repositories. These repositories not only store the relevant data but also illustrate the evolution of each modeling tool under study through structured commits and project organization.

One GitHub repository[12] is dedicated to MetaEdit+. It is structured such that each commit corresponds to a specific evolution scenario as outlined in our paper. This chronological arrangement allows researchers and practitioners to follow the sequence of changes applied during the co-evolution process, providing a clear view of how each scenario impacts the tool and the models.

Similarly, we have published the initial implementation of Gothic with EMF/Sirius in a single GitHub repository.[13] For EMF/Sirius, the structure mirrors that of the MetaEdit+ repository, with each commit representing a distinct step in the co-evolution scenarios.

Since Jjodel does not support versioning natively, we published the implementation of the Gothic versions to a public installation.[14] There too, each scenario is captured as a separate project where the evaluators have manually performed and documented the changes, allowing for a clear comparison and analysis despite the absence of built-in version control.

In addition to the three version repositories, we have also published the initial version of each state machine implementation to Zenodo [52], along with the Supplementary Material. While this Zenodo entry provides a snapshot of the repository at a specific point in time, it does not allow the interested reader to inspect the complete commit history, which is a crucial element in fully understanding the scope and depth of our contributions. Therefore, while the Zenodo entry serves as a valuable static reference, the GitHub repositories are recommended for those interested in a dynamic and detailed examination of the evolutionary changes.

By presenting the evolution of these tools in this structured fashion, we aim to provide a comprehensive resource for understanding and analyzing the co-evolutionary capabilities of different modeling tools. We encourage other researchers to utilize these repositories not only to validate our findings but also to conduct further investigations into the co-evolution of modeling languages and tools.

[12] https://github.com/mccjpt/Gothic.

[13] https://github.com/MDEGroup/coevolution-impact-analysis.

[14] https://mdegroup.github.io/coevolution-impact-analysis/.

8.2 Applicability of the tool evaluation framework

Applying the evaluation framework successfully to assess three significantly different tools indicates that it is viable. It is also functional as it shows clear differences among the tools' capabilities to cope with co-evolution. The tools' scores covered the whole spectrum of evaluation, ranging from 1 to 5. Although not every possible score was present in the final results, all were present at some point, before cross-checking produced the final scenario implementations and scores.

The scenario for renaming constraints was not relevant in MetaEdit+ and Sirius here, as the constraints used in these scenarios do not have names, nor for Jjodel, which currently does not support that kind of constraint. Some constraint implementations in these tools would have had names, and other tools are known to have named constraints, e.g., in constraint languages. This may be a good indication of the benefits of deriving the scenarios from first principles, rather than tailoring them to selected tools.

The task for scenario 10 was hard to implement in Sirius and Jjodel, as they use the EMF metametamodel, which does not directly support the concept of a role (one end of a relationship). A replacement task was found, albeit somewhat different: more of a refactoring on the technical level than a change in the language, and requiring no model co-evolution rather than the intensive model co-evolution of the original task.

All tools evaluated performed perfectly when the location of change was the notation. This seems to fit with the fact that notation generally depends on the metamodel, rather than vice versa. Another reason for this could be the rather simple visualization used by the Gothic Security language. More complicated visualizations that reuse notation elements could see a change in one notation element result in a problem in another notation element.

Clearly, the evaluation framework is significantly more stringent than earlier frameworks, covering more aspects and over the whole extent of the language, tools, generators and models. This allowed us to find cases with room for improvement in tools like MetaEdit+, as opposed to its full marks on earlier evaluation frameworks in Table 1.

The effort to perform the co-evolution scenarios with each tool was rather modest: even in the slowest case it took under an hour. This indicates that the cost to evaluate other tools is likely to be modest too, enabling others to relatively easily repeat this evaluation.

8.3 Threats to validity

When analyzing and comparing the three tools, there are potential threats to validity represented by unconscious bias because the authors are involved in designing two of the tools.

We might unintentionally favor our tools in terms of how we present their features, how we interpret results, or in the comparison methodology. This risk has been mitigated by having the results checked by a different person from the one who executed the experiment, and by all authors checking and agreeing the resulting scores.

The main threats to validity of a new evaluation framework are whether it can be applied to give consistent results for a given tool and whether it offers a valuable comparison across different tools. Our comparison is based on a specific set of scenarios, and the results might not apply to other contexts. Having to change a scenario for specific tools makes scores and times less comparable. However, as we built on an earlier framework, a more comprehensive application could also improve and verify our framework. We thus invite others to repeat the evaluation described here and to apply it to evaluate other tools. More extensive cases are also welcome, along with others' experience of industrial-scale use.

Assigning scores based on descriptions may be affected by interpretation or subjective factors. This was mitigated by using scores based on observable facts rather than opinions, and by producing subscore-specific versions of the generic scoring criteria (see Supplementary Material [52]). The timings will depend to some extent on the skill and speed of the person performing the evaluation. This is mitigated to some extent by choosing people experienced with each tool. Executing the scenarios with MetaEdit+ obtained similar times on each of three repetitions. Even for the same person and tool, however, significant differences in times would be seen for different priorities: e.g. better deprecation guidance for modelers or faster execution time.

8.4 Final remarks

From our analysis, several key insights have emerged regarding the tools utilized in model-driven engineering, mainly focusing on automating co-evolution tasks. Firstly, the tools can be broadly categorized into integrated environments like MetaEdit+ and Jjodel and component-based systems like EMF/Sirius. Integrated environments offer a significant advantage in automating co-evolution tasks, especially when coupled with reflectiveness [66], i.e., the ability to self-reflect and adapt to changes in the environment. Such capabilities can typically provide native support for consistency management by fully exploiting the dependencies underlying the meta-architecture in model-driven tools. This cohesive integration enables a more seamless adaptation process than component-based systems, where each co-evolution task requires bespoke design and implementation to manage dependencies effectively. Additionally, solutions developed with EMF/Sirius often rely on components created by various organizations or individual developers, potentially leading to less efficient integration due to the disparate nature of these contributions. Secondly, our findings highlight that the effectiveness of automated methods for co-evolution significantly increases when the artifacts adapted in response to a metamodel change are models themselves or when the artifact, such as a code generation template, employs a model that abstracts from irrelevant details and provides an intermediate representation. This use of models facilitates a more precise representation of the dependencies between the metamodel and the artifacts under study, thereby enhancing the adaptability and consistency of the co-evolution process.

Moreover, the manner in which tools such as Jjodel are used, particularly the choice of identification methods, significantly influences co-evolution activities. Within Jjodel, we referenced metamodel elements by internal ID in generators, but by name in graphical notation definitions. The choice of which approach to use dramatically affects the resilience and adaptability of its ecosystem during co-evolution processes. These identification methods, alongside other operational settings and referencing styles, play a crucial role in determining the outcome of co-evolutionary activities. For a comprehensive discussion on how these factors interact and impact co-evolution within the Jjodel environment, please see Sect. 7.2.

As mentioned earlier, our experience suggests that where language changes such as removals reduce the set of valid models, organizations prefer an approach based on deprecation rather than strictly formalist attempts to automatically or semi-automatically migrate models wholesale. Conversely, where a change like adding a new language element can indeed be applied automatically with no danger, a completely automatic and largely invisible approach is preferred.

Using deprecation like this follows experience with other languages such as those for programming. Deprecation is also often seen in libraries and APIs, where, as with languages, there are few definers and many users. It is also analogous to a familiar process in natural language: the linguistic concept of a grammatical construct ceasing to be productive, i.e. no longer being used to form new material. When a construct is no longer productive, existing material does not disappear but remains without changing. For instance, the old plural ending *-en* in English is no longer used to form new plurals, but old forms like child*ren* and ox*en* are still in use.

Making co-evolution fully automatic in cases where it is always known to work, based on the location and nature of the change, is clearly a positive factor in maintaining sustainability with language evolution. For the remaining cases, deprecation seems to be the strategy favored in industrial use—even where automation facilities are provided. As such facilities will be needed relatively rarely, it seems best to offer them in a familiar programming language, rather than require learning a new more domain-specific language.

9 Conclusions

We presented a framework to evaluate tool support for co-evolution of languages and models made with them. The framework builds on and combines previous work, and makes it more stringent. It covers changes in language constraints and concrete syntax as well as in the abstract syntax, and the impact on all parts of the language and generators, as well as modeling tools and models. Its scoring offers a more nuanced scale, and hopefully one which is easier to apply consistently across different parts of the modeling ecosystem and different tools.

The evaluation shows that in MetaEdit+ and Jjodel, editors do not break and existing models continue to work, largely avoiding the need to create transformations to co-evolve models. The majority of updates are achieved through automatic, built-in co-evolution of models and modeling tools. In contrast, all changes to the metamodel or constraints in EMF/Sirius resulted in significant problems, with tool errors and often complicated manual intervention needed to continue working. In over half of those cases, the errors also required manual work in models.

These differences do not seem to be explained by traditional distinctions between tools: industry/academic, open source/closed source, mature/research, commercial/free. As both MetaEdit+ and Jjodel have had strong co-evolution support from the start, and EMF/Sirius has not improved significantly despite problems being noticed early on, it seems that the cause may be architectural, and thus hard to change via new versions. One hypothesis is the more integrated nature of MetaEdit+ and Jjodel tools and development teams, compared with the more separate components and teams of EMF/Sirius. Another hypothesis is the persistence formats, with MetaEdit+ and Jjodel staying closer to the in-memory structure of objects and keeping all artifacts integrated, whereas Sirius splits metamodels and models into several different types of loosely linked XML files.

MetaEdit+ made the modeler's life simpler by following the established practice of deprecation rather than deletion. Although the choice to provide detailed deprecation guidance required some work, the overall time indicates that the work is a manageable burden. Jjodel and EMF/Sirius do not provide any means for managing deprecation; the problem is partly mitigated in Jjodel by the possibility of untyping and retyping model elements.

In Jjodel, the overall time of 10 min shows the benefits of its automated model adaptation and general resilience, although it was unable to accomplish three of the 12 scenarios, with another one unnecessary. Including deprecation guidance, MetaEdit+ had an overall time of 32 min (one scenario unnecessary). EMF/Sirius required an overall time of 56 min (two scenarios unnecessary), and with the lowest possible score for model co-evolution on four scenarios, the time on larger models would grow much faster than in the other tools.

While our evaluation framework addresses a wider range of co-evolution than previous research, there is still plenty of room for future work. The evaluation of tool capabilities could be expanded from the current single language and model to encompass multiple integrated languages and models. Changes in one language may necessitate corresponding changes in other languages and their associated tools or toolchains. Another direction for future research is extending the evaluation framework to consider collaboration: multiple language engineers making changes while multiple modelers work on models. A third direction would be addressing longer term use on an industrial scale, where language workbenches themselves have new versions, possibly with evolution of their metametamodel and language definition capabilities.

Acknowledgements Part of this work has been supported by the EMELIOT national research project, which has been funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY). We acknowledge the Italian "PRIN 2022" project TRex-SE: "Trustworthy Recommenders for Software Engineers, Project ID: 2022LKJWHC."

Funding Open Access funding provided by University of Jyväskylä (JYU).

References

1. Sprinkle, J., Mernik, M., Tolvanen, J.-P., Spinellis, D.: What kinds of nails need a domain-specific hammer?: domain-specific modeling. IEEE Softw. **26**(4), 15–18 (2009)
2. Kelly, S., Pohjonen, R.: Worst practices for domain-specific modeling. IEEE Softw. **26**(4), 22–29 (2009)
3. Hebig, R., Khelladi, D.E., Bendraou, R.: Approaches to co-evolution of metamodels and models: a survey. IEEE Trans. Softw. Eng. **43**(5), 396–414 (2016)
4. Borum, H.S., Seidl, C.: Survey of established practices in the life cycle of domain-specific languages. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, pp. 266–277 (2022)
5. Lago, P., Koçak, S.A., Crnkovic, I., Penzenstadler, B.: Framing sustainability as a property of software quality. Commun. ACM **58**(10), 70–78 (2015)
6. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial adoption of model-driven engineering: Are the tools

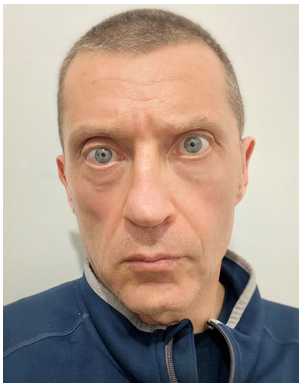
really the problem? In: Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Proceedings 16, pp. 1–17. Springer, Miami, FL, USA (2013),
7. DSMForum, http://dsmforum.org/cases.html (accessed april 2023) (2023)
8. Fowler, Martin: Domain-Specific Languages. Pearson Education (2010)
9. Erdweg, S., Van Der Storm, T., Völter, M., Boersma, M., Bosman, R., R Cook, W., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A. et al.: The state of the art in language workbenches: conclusions from the language workbench challenge. In: Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA. Proceedings 6, pp. 197–217. Springer (2013)
10. El Kouhen, A., Dumoulin, C., Gerard, S., Boulet, P.: Evaluation of modeling tools adaptation. Technical report, CNRS HAL hal-00706701. URL http://tinyurl.com/gerard12 (2012)
11. Tolvanen, J.-P., Kelly, S.: Effort used to create domain-specific modeling languages. In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 235–244 (2018)
12. Di Ruscio, D., Lämmel, R., Pierantonio, A.: Automated co-evolution of GMF editor models. In: Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers 3, pp. 143–162. Springer (2011)
13. Di Rocco, J., Di Ruscio, D., Narayanankutty, H., Pierantonio, A.: Resilience in Sirius editors: understanding the impact of metamodel changes. In MoDELS (Workshops), pp. 620–630 (2018)
14. Tolvanen, J.-P., Kelly, S.: Evaluating tool support for co-evolution of modeling languages, tools and models. In: Workshop on Models and Evolution (ME 2023), MoDELS 2023 Companion, pp. 914–923. IEEE Computer Society, Los Alamitos, CA, USA (2023)
15. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+: a fully configurable multi-user and multi-tool CASE and CAME environment. In: Advanced Information Systems Engineering: 8th International Conference, CAiSE'96 Heraklion, Crete, Greece, Proceedings 8, pp. 1–21. Springer (1996)
16. Eclipse, Sirius 7.3: Documentation https://https://eclipse.dev/sirius/doc/ (2023) Accessed January 2024
17. Di Vincenzo, D., Di Rocco, J., Di Ruscio, D., Pierantonio, A.: Enhancing syntax expressiveness in domain-specific modelling. In: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 586–594. IEEE (2021)
18. Iovino, L., Rutle, A., Pierantonio, A., Di Rocco, J.: Query-based impact analysis of metamodel evolutions. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 458–465 (2019). https://doi.org/10.1109/SEAA.2019.00074
19. Meyers, B., Vangheluwe, H.: A framework for evolution of modelling languages. Sci. Comput. Program. **76**(12), 1223–1246 (2011)
20. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: European Conference on Object-Oriented Programming, pp. 600–624. Springer (2007)
21. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, pp. 164–173 (2007)
22. Gruschko, B., Kolovos, D., Paige, R.: Towards synchronizing models with evolving metamodels. In: Proceedings of the International Workshop on Model-Driven Software Evolution, p. 3. Amsterdam, The Netherlands (2007)
23. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: 2008 12th International IEEE Enterprise Distributed Object Computing Conference, pp. 222–231. IEEE (2008)
24. Narayanan, A., Levendovszky, T., Balasubramanian, D., Karsai, G.: Automatic domain model migration to manage metamodel evolution. In: Schürr, Andy, Selic, Bran (eds.) Model Driven Engineering Languages and Systems, pp. 706–711. Springer, Heidelberg, Berlin (2009)
25. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE—automating coupled evolution of metamodels and models. In: European Conference on Object-Oriented Programming, pp. 52–76. Springer (2009)
26. Herrmannsdörfer, M.: Evolutionary Metamodeling. PhD thesis, Technical University of Munich (2011)
27. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model migration with Epsilon Flock. In: Theory and Practice of Model Transformations: Third International Conference, ICMT 2010, Malaga, Spain. Proceedings 3, pp. 184–198. Springer (2010)
28. Kessentini, W., Sahraoui, H., Wimmer, M.: Automated metamodel/model co-evolution: a search-based approach. Inf. Softw. Technol. **106**, 49–67 (2019)
29. Barriga, A., Rutle, A., Heldal, R.: Personalized and automatic model repairing using reinforcement learning. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 175–181 (2019). https://doi.org/10.1109/MODELS-C.2019.00030
30. García, J., Diaz, O., Azanza, M.: Model transformation co-evolution: a semi-automatic approach. In: International Conference on Software Language Engineering, pp. 144–163. Springer (2012)
31. Di Ruscio, D., Iovino, L., Pierantonio, A.: A methodological approach for the coupled evolution of metamodels and ATL transformations. In: International Conference on Theory and Practice of Model Transformations, pp. 60–75. Springer (2013)
32. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Dealing with the coupled evolution of metamodels and model-to-text transformations. In: Me@ Models, pp. 22–31. Citeseer (2014)
33. Demuth, A., Lopez-Herrejon, R.E., Egyed, A.: Supporting the co-evolution of metamodels and constraints through incremental constraint management. In: Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA. Proceedings 16, pp. 287–303. Springer (2013)
34. Batot, E., Kessentini, W., Sahraoui, H., Famelis, M.: Heuristic-based recommendation for metamodel-OCL coevolution. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 210–220. IEEE (2017)
35. Kusel, Angelika, Etzlstorfer, Juergen, Kapsammer, Elisabeth, Retschitzegger, Werner, Schoenboeck, Johannes, Schwinger, Wieland, Wimmer, Manuel: Systematic co-evolution of OCL expressions. 11th APCCM **27**, 30 (2015)
36. Kessentini, W., Wimmer, M., Sahraoui, H.: Integrating the designer in-the-loop for metamodel/model co-evolution via interactive computational search. In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 101–111 (2018a)
37. Di Ruscio, D., Etzlstorfer, J., Iovino, L., Pierantonio, A., Schwinger, W.: A feature-based approach for variability exploration and resolution in model transformation migration. In: European Conference on Modelling Foundations and Applications, pp. 71–89. Springer (2017)
38. Stevens, P.: Connecting software build with maintaining consistency between models: towards sound, optimal, and flexible building from megamodels. Softw. Syst. Model. **19**(4), 935–958 (2020)
39. Bézivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: Proceedings of the OOPSLA/GPCE: Best Practices for Model-driven Software Development Workshop, 19th Annual

ACM Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 1–9. Citeseer (2004)
40. Ozkaya, M., Erata, F.: Understanding practitioners' challenges on software modeling: a survey. J. Comput. Lang. **58**, 100963 (2020)
41. OMG, Systems Modeling Language: SysML v1 to SysML v2 Transformation, Release 2023-02, 2023 (2023)
42. Ozkaya, M., Akdur, D.: What do practitioners expect from the meta-modeling tools? A survey. J. Comput. Lang. **63**, 101030 (2021)
43. Schuts, M., Alonso, M., Hooman, J.: Industrial experiences with the evolution of a DSL. In: Proceedings of the 18th ACM SIGPLAN International Workshop on Domain-Specific Modeling, pp. 21–30 (2021)
44. Mengerink, J.: The DSL/Model co-evolution problem in industrial MDE ecosystems. Phd Dissertation, Eindhoven University of Technology 2018 (2018)
45. Akesson, B., Hooman, J., Sleuters, J., Yankov, A.: Reducing design time and promoting evolvability using domain-specific languages in an industrial context. In: Model Management and Analytics for Large Scale Systems, pp. 245–272. Elsevier (2020)
46. de Geest, G., Savelkoul, A., Alikoski, A.: Building a framework to support domain-specific language evolution using Microsoft DSL Tools. In: Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modelling, p. 4. Citeseer (2007)
47. Ratiu, D., Nehls, H., Joanni, A., Rothbauer, S.: Use MPS to unleash the creativity of domain experts: language engineering is a key enabler for bringing innovation in industry. In: Domain-Specific Languages in Practice: with JetBrains MPS, pp. 25–52. Springer (2021)
48. Kelly, S., Tolvanen, J.-P.: Automated annotations in domain-specific models: analysis of 23 cases. In: STAF Workshops, pp. 77–87 (2021a)
49. Kelly, S., Rossi, M., Tolvanen, J.-P.: What is needed in a metaCASE environment? Enterp. Model. Inf. Syst. Archit. (EMISAJ) **1**(1), 25–35 (2005)
50. Steven, K., Juha-Pekka, T.: Domain-Specific Modeling: Enabling Full Code Generation. IEEE Press, Wiley (2008)
51. Iovino, L., Di Salle, A., Di Ruscio, D., Pierantonio, A.: Metamodel deprecation to manage technical debt in model co-evolution. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–10 (2020)
52. Tolvanen, J.-P., Kelly, S., Di Rocco, J., Pierantonio, A., Tinella, G.: Replication package and supplementary material for: a framework for evaluating tool support for co-evolution of modeling languages. Tools Models (2024). https://doi.org/10.5281/zenodo.12163923
53. Cooper, J., De la Vega, A., Paige, R., Kolovos, D., Bennett, M., Brown, C., Pina, B.S., Rodriguez, H.H.: Model-based development of engine control systems: experiences and lessons learnt. In: 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 308–319. IEEE (2021)
54. Sghaier, O.B., Sahraoui, H., Famelis, M.: Metamodel refactoring using constraint solving: a quality-based perspective. In: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 797–806. IEEE (2021)
55. Kelly, S.: Empirical comparison of language workbenches. In: Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling, DSM '13, pp. 33–38. Association for Computing Machinery. New York, NY, USA (2013). https://doi.org/10.1145/2541928.2541935
56. MetaCase: MetaEdit+ 5.5 User's Guides. https://metacase.com/support/55/manuals/. Accessed January 2024 (2023)
57. Kelly, S., Tolvanen, J.-P.: Collaborative modelling and metamodelling with MetaEdit+. In: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 27–34. IEEE (2021b)
58. Tolvanen, J.-P., Kelly, S.: Model-driven development challenges and solutions: experiences with domain-specific modelling in industry. In: 2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 711–719. IEEE (2016)
59. Kelly, Steven, Lyytinen, Kalle, Rossi, Matti, Tolvanen, Juha Pekka: MetaEdit+ at the age of 20. In: Bubenko, Janis, Krogstie, John, Pastor, Oscar, Pernici, Barbara, Rolland, Colette, Sølvberg, Arne (eds.) Seminal Contributions to Information Systems Engineering: 25 Years of CAiSE, pp. 131–137. Springer, Heidelberg, Berlin (2013). https://doi.org/10.1007/978-3-642-36926-1_10
60. Kessentini, W., Wimmer, M., Sahraoui, H.: Integrating the designer in-the-loop for metamodel/model co-evolution via interactive computational search. In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '18, pp. 101-111. New York, NY, USA, Association for Computing Machinery (2018b). https://doi.org/10.1145/3239372.3239375
61. Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schwinger, W., Schönböck, J.: Consistent co-evolution of models and transformations. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 116–125 (2015b). https://doi.org/10.1109/MODELS.2015.7338242
62. Khelladi, D.E., Bendraou, R., Hebig, R., Gervais, M.P.: A semi-automatic maintenance and co-evolution of OCL constraints with (meta)model evolution. J. Syst. Softw. **134**, 242–260 (2017). https://doi.org/10.1016/j.jss.2017.09.010
63. Zhang, W., Hebig, R., Strüber, D., Steghöfer, J.P.: Automated extraction of grammar optimization rule configurations for metamodel-grammar co-evolution. In: Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2023, pp. 84–96. New York, NY, USA, Association for Computing Machinery (2023). https://doi.org/10.1145/3623476.3623525
64. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The epsilon generation language. In: Schieferdecker, Ina, Hartman, Alan (eds.) Model Driven Architecture-Foundations and Applications, pp. 1–16. Springer, Berlin, Heidelberg (2008)
65. Efftinge, S., Friese, P., Hase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein, J., Moroff, D., Thoms, K., Völter, M., et al.: Xpand Documentation. Eclipse Foundation, Tech. Rep., Ottawa, Canada (2004)
66. Maes, P.: Concepts and experiments in computational reflection. ACM Sigplan Not. **22**(12), 147–155 (1987)

Juha-Pekka Tolvanen is the CEO of MetaCase and holds the position of an adjunct professor at the University of Jyväskylä. He acts as a consultant worldwide for modeling language and code generation development. Juha-Pekka has been involved in domain-specific approaches and tools, notably metamodeling, modeling and code generator development since 1991. He has a PhD in computer science from the University of Jyväskylä.

Steven Kelly is the CTO of MetaCase and co-founder of the DSM Forum. As architect and lead developer of MetaEdit+, he has over thirty years of experience of building language workbenches and modeling tools. He has worked as a DSM consultant to over 100 clients around the world, is co-author of a book on domain-specific modeling and has published over 70 articles. Steven has a PhD in computer science from the University of Jyväskylä and a master's degree from Cambridge.

Juri Di Rocco is a tenure-track assistant professor in the Department of Information Engineering Computer Science and Mathematics at the University of L'Aquila. He earned his PhD from the same institution as part of the MDE-Group research team, under the supervision of Alfonso Pierantonio and Davide Di Ruscio. His research focuses on various aspects of software language engineering, with particular interest in model-driven engineering (MDE). His key areas of study include domain-specific modeling languages, recommender systems for MDE, model repositories, and mining techniques.



Alfonso Pierantonio is a Professor of Software Engineering at the University of L'Aquila, Italy. He specializes in model-driven and language engineering, with a strong interest in co-evolution techniques, consistency management, and tool design and implementation. Alfonso is Editor-in-Chief of the Journal of Object Technology and on the editorial and advisory board of Software and System Modeling and Science of Computer Programming. He is in the Steering Committee of ACM/IEEE MoDELS.

Giordano Tinella is currently a master's student in Advanced Software Engineering and is set to begin a PhD at the University of L'Aquila (UNIVAQ). His research centers on software language engineering, with a strong focus on model-driven engineering (MDE). His main areas of expertise include collaborative modeling, co-evolution, models@runtime, and digital twins.