ARTICLE IN PRESS

Science of Computer Programming ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

# A framework for evolution of modelling languages

Bart Meyers [a], Hans Vangheluwe [a,b,*]

[a] Modelling, Simulation and Design Lab (MSDL), University of Antwerp, Middelheimlaan 1, B-2020 Antwerp, Belgium
[b] Modelling, Simulation and Design Lab (MSDL), McGill University, 3480 University Street, H3A 2A7 Montréal, Québec, Canada

ARTICLE INFO

*Article history:*
Available online xxxx

*Keywords:*
Evolution
Modelling languages
Language engineering
Model-driven engineering
Model transformation

ABSTRACT

In model-driven engineering, evolution is inevitable over the course of the complete life cycle of complex software-intensive systems and more importantly of entire product families. Not only instance models, but also entire modelling languages are subject to change. This is in particular true for domain-specific languages, whose language constructs are tightly coupled to an application domain.
The most popular approach to evolution in the modelling domain is a manual process, with tedious and error-prone migration of artefacts such as instance models as a result. This paper provides a taxonomy for evolution of modelling languages and discusses the different evolution scenarios for various kinds of modelling artefacts, such as instance models, meta-models, and transformation models. Subsequently, the consequences of evolution and the required remedial actions are decomposed into primitive scenarios such that all possible evolutions can be covered exhaustively. These primitives are then used in a high-level framework for the evolution of modelling languages.
We suggest that our structured approach enables the design of (semi-)automatic modelling language evolution solutions.



## 1. Introduction

The systems we analyse and design are characterised by an ever increasing complexity. As demands on quality grow, development and production cost must still be kept minimal. A recent approach to tackle this complexity is Model-Driven Engineering (MDE) [44]. A system is described using the most appropriate modelling language(s), at the most appropriate level(s) of abstraction [32]. The goal is to minimise "accidental" (as opposed to "essential") complexity [3] by capturing only the essence of a problem. More specifically, in Domain-Specific Modelling (DSM) [19], expertise of a domain expert such as a mechatronics engineer or a hospital process manager is encoded in the form of models in Domain-Specific Modelling Languages (DSMLs), which separate problem domain knowledge from implementation target (software and/or hardware) domain knowledge. When the syntax and semantics of DSMLs are precisely defined (by means of meta-modelling [21] and model transformation [7]), models can be used for analysis, simulation and even full code synthesis. Often, multiple aspects/views as well as sub-systems are modelled using distinct DSMLs. Anecdotal evidence shows five to ten times productivity increases because of domain-specific modelling [19].

In software engineering, the evolution of software artefacts is ubiquitous [28]. Diverse artefacts such as programs, data, requirements, and documentation may evolve. In MDE, where modelling languages play a central role, evolution occurs not only at the level of models, but also at the level of modelling languages. This is in contrast with general-purpose programming languages such as C++ where programs evolve, but not the programming language. Language evolution applies in particular

* Corresponding author at: Modelling, Simulation and Design Lab (MSDL), University of Antwerp, Middelheimlaan 1, B-2020 Antwerp, Belgium.
*E-mail addresses:* Bart.Meyers@ua.ac.be (B. Meyers), Hans.Vangheluwe@ua.ac.be, hv@cs.mcgill.ca (H. Vangheluwe).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.01.002

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

to DSMLs, where relatively frequent changes in the problem domain as well as in the implementation target domain (*e.g.,* due to external technical or strategic decisions) must be reflected in the respective languages. This is to maintain the high coupling between domain and language. The first problem is the need for rapid development techniques for DSMLs, as they are created and modified frequently during the life cycle of the system they are used for. The second, and far greater problem is that possibly large numbers of modelling artefacts such as instance models or transformation models developed become invalid and unusable when a related DSML is modified/evolved. Early adopters of MDE and DSM dealt with language evolution issues manually [43]. However, this approach, as well as an ad hoc approach to any language change, is tedious and error-prone [49]. The reason for this is that syntax of languages such as UML [37] and BPMN [35], which have evolved considerably over the last few years, easily comprise several hundreds of elements. Also, the semantic differences resulting from this evolution, either intended or intentional, can be subtle. Hence, dealing with evolution requires in-depth knowledge of the language as a whole. Without a proper scientific foundation, as well as methods, techniques and tools to support evolution, MDE in general and DSM in particular, cannot live up to its promise of ten-fold productivity increase [19]. This becomes apparent when projects span longer periods of time [43]. Since the problem of modelling language evolution was first identified by Sprinkle and Karsai [47], the general problem has only grown in importance, yet still remains largely unsolved. The importance of modelling language evolution is further evidenced by the attention it receives in the research community. The evolution of modelling languages is one of the 11 topics for paper submission at MODELS 2010 (ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems), and workshops such as ME 2010 (International Workshop on Models and Evolution) are devoted largely to the topic. Current state-of-the-art contributions in this field are focused on (semi-)automatic model differencing [6] and on the co-evolution of instance models [16].

The remainder of the paper is organised as follows: Section 2 is a short introduction to modelling languages. Section 3 discusses related work. Section 4 introduces an example that will be used to illustrate our approach throughout the paper. Section 5 presents the possible kinds of evolution. Section 6 introduces a way to tackle evolution of modelling languages by deconstructing the problem into primitives. Section 7 presents a framework and algorithm for the evolution of modelling artefacts when languages evolve. Section 8 concludes the paper and describes future work.

## 2. Modelling languages

To allow for a precise discussion of language evolution, we briefly introduce fundamental modelling language concepts. This introduction which we elaborated in [10] is based on foundations laid by Harel and Rumpe [13] and Kühne [21]. The two main aspects of a model are its *syntax* (how it is represented) and its *semantics* (what it means).

Firstly, the syntax comprises *concrete syntax* and *abstract syntax*. The concrete syntax describes how the model is represented (*e.g.,* in 2D vector graphics or in textual form), which can be used for model input as well as for model visualisation. The abstract syntax contains the "essence" of the model (*e.g.,* as a typed Abstract Syntax Graph (ASG)—when models are represented as graphs).

A single abstract syntax may be represented by multiple concrete syntaxes. There exists a mapping between a concrete syntax and its abstract syntax, called the *parsing function*. There is also a mapping in the opposite direction, called the *rendering function*. These are the *concrete mapping functions*. Mappings are usually implemented, or can at least be represented, as model transformations. The abstract syntax and concrete syntax of a model are related by a surjective homomorphic function that translates a concrete syntax graph into an abstract syntax graph.

Secondly, the semantics of a model are defined by a complete, total and unique *semantic mapping function* which maps every abstract syntax model onto a single element in a *semantic domain*, such as Ordinary Differential Equations, Petri nets [39], or a set of behaviour traces. These are domains with well-known and precise semantics. For convenience, semantic mapping is usually performed on abstract syntax, rather than on concrete syntax directly. More explicitly, the abstract syntax can be used as a basis for semantic anchoring [4].

A meta-model is a finite model that explicitly describes the abstract syntax and static semantics, which are statically checkable, of a language. Dynamic semantics are not covered by the meta-model. The abstract syntax of a model can be represented as a graph, where the nodes are elements of the language and the edges are relations between these elements, and also elements of the language. Instance models of the language are said to *conform to* the meta-model of the language. In [21], Kühne refers to this relation as *linguistic instance of*. The description of the abstract syntax is typically specified in a modelling language such as UML Class Diagrams [34]. Static semantics can be described in a constraint language such as the Object Constraint Language (OCL) [36]). Often, but not necessarily, the concrete syntax mapping is directly attached to a meta-model, where every element of the concrete syntax can be explicitly traced back to its corresponding element of the abstract syntax.

Fig. 1 shows the different kinds of relations involving a model $m$. Relations are visualised by arrows, "conforms to"-relationships are dotted arrows. The abstract syntax model $m$ conforms to a meta-model $MM_{Lang}$, the explicit model of the language *Lang*. There is a rendering function $\kappa_i$ between $m$ and a concrete syntax $\kappa_i(m)$ model. The inverse of $\kappa_i$ is a parsing function $\pi_i$ so that $\pi_i(\kappa_i(m)) = m$. The index $i$ highlights the fact that multiple concrete representations may be used. $\kappa_i(m)$ conforms to a meta-model $MM_{CS\_\kappa_i}$, the explicit model of the concrete syntax language (such as the set of all 2D vector graphics drawings). Semantics are described by the semantic mapping function *[[.]]*, and maps $m$ to a model *[[m]]* in the semantic domain. This semantic domain is a different modelling language with its own syntax en semantics. Similar to $m$ conforming to $MM_{Lang}$, *[[m]]* conforms to $MM_{SemDom}$. Additionally, transformations $T_j$ may be defined for $m$.

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

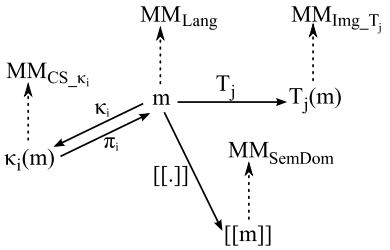


**Fig. 1.** A model and its relations in the context of MDE.

The index *j* highlights the fact that multiple general transformations (*e.g.,* for synthesis, migration, abstraction/refinement, normalisation, optimisation, etc. [29]) may exist.

Assuming that a modelled system consists of models in an appropriate language and explicit relations between these models, we posit that the architecture of any modelled system can be mapped on (multiple instances of) Fig. 1. Multiple models *m* can exist in a system, typically all conforming to the same meta-model $MM_{Lang}$. Each model can be mapped on Fig. 1 separately. Note that $T_j(m)$ or *[[m]]* can themselves be seen as a model *m* in Fig. 1, to which possibly relations such as *T* and *[[.]]* apply. In domain-specific modelling, when languages are explicitly modelled the meta-model $MM_{Lang}$ of a domain-specific language (DSL) is an artefact that is part of the system in its own right. Thus, $MM_{Lang}$ itself can be seen as an *m* in Fig. 1, having a meta-model in its own right. Note that programs can also be considered as models, with an abstract syntax tree, a concrete syntax and a semantic mapping to, for instance, machine code or an operational semantics in the form of an interpreter.

In conclusion, Fig. 1 describes any explicitly modelled system. Although the statement that all possible systems can be mapped to the above diagram cannot be formally proved, we are confident that it holds, based on our experience with modelling language engineering, in particular with AToM$^3$ (A Tool for Multi-formalism and Meta-Modelling) [7]. From now on, we assume Fig. 1 describes any explicitly modelled system, and hence a framework for evolution must support the possible scenarios that emerge from it.

## 3. Related work

In this section, work related to evolution is presented and some useful concepts are introduced. This mainly covers the related topics of model differencing and model co-evolution, on which we will build.

### 3.1. Model differencing

In order to be able to model evolution in-the-large, one should be able to model differences between two versions of a model. This can of course be done by using lexical differencing, as used for text files, on the data representation of the model. However, the result of such analysis is often not useful, as (1) the actual differences occur at the granularity level of nodes, links, labels and attributes and (2) models are usually not sequential in nature and equivalences between models will not be taken into account. Hence, model differencing should be done at an appropriate level of abstraction, and take semantics into account. Progress has been made in this area [1,6,27,38,55]. Existing approaches typically rely on the abstract syntax graphs (ASGs) of the two models to compare, and traverse both graphs in parallel. Nodes in the graphs are matched by matching unique identifiers [1,38], or by a number of heuristics [27,55]. However, no comprehensive approach that computes the differences between graph-like models exists yet. As a direct result, no general model version control system exists today.

In addition to finding differences, one should be able to *represent* them explicitly as a model, called the *delta model*. There are two kinds of representations: operational and structural. In the operational (or change-based) representation, the difference between two versions of a model is modelled as the series of CRUD (Create/Read/Update/Delete) edit operations that were performed on one model to arrive at the other [1,16]. When these operations are recorded live from a tool, this strategy is very accurate and powerful, though dependent on that particular tool and difficult to manipulate explicitly. In structural (or state-based) representations, either the model is coloured [27,38,45,55] or a designated delta model is created which can be used by modelling tools as yet another model in an appropriate language [6,47].

### 3.2. Model co-evolution

When the syntax of a modelling language evolves (*i.e.,* the meta-model evolves), the most prominent side effect is that its instance models may no longer conform to the new meta-model. Therefore, the co-evolution (with evolution of their meta-model) of models has become an important research topic. This research is inspired by the way the problem of language evolution is identified or dealt with in other domains. In grammar evolution [20,23,40], the type/instance relation is analogous to the meta-model element/instance relation. This type/instance relation is also present in programming languages. In database schema evolution, database tables have to be migrated after a change in the database schema [2,26,41]. In format evolution, formally specified documents (*e.g.,* XML documents) must be migrated when their format (*e.g.,* specified in a DTD) changes [24,48].

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

**Table 1**
Delta operations based on [5], with their migration operations and inverse operations.

| Delta operation | Type | Migration operation | Inverse operation |
|---|---|---|---|
| Non-breaking delta operations | | | |
| Generalise meta-property | Additive | None | Restrict meta-property |
| Add non-obligatory meta-class | Additive | None | Eliminate meta-class |
| Add non-obligatory meta-property | Additive | None | Eliminate meta-property |
| Extract superclass | Additive | None | Flatten hierarchy |
| Breaking and resolvable delta operations | | | |
| Eliminate meta-class | Subtractive | Eliminate instances | Add non-obligatory meta-class |
| Eliminate meta-property | Subtractive | Eliminate instances | Add non-obligatory meta-property |
| Push meta-property | Subtractive | Eliminate properties from superclass instances | Pull meta-property |
| Flatten hierarchy | Subtractive | Eliminate superclass instances | Extract superclass |
| Rename meta-class | Updative | Change instances | Rename meta-element |
| Rename meta-property | Updative | Change instances | Rename meta-element |
| Breaking and unresolvable delta operations | | | |
| Add obligatory meta-class | Additive | Add default instances | Eliminate meta-class |
| Add obligatory meta-property | Additive | Add default instances | Eliminate meta-property |
| Pull meta-property | Additive | Add default properties for superclass instances | Push meta-property |
| Restrict meta-property | Subtractive | Remove instance if non-compliant | Generalise meta-property |

It is widely accepted that a model co-evolution (*i.e.,* migration) is best modelled as a model transformation [5,11,15–17,47,50,51,56], which we will call the *migration transformation*. Gruschko et al. write this transformation manually using the Epsilon Transformation Language (ETL) [11].

Most approaches define some specific operations as building blocks for evolution, similar to the operational representation of model differences. Such operations typically include "create meta-class", "restrict multiplicity on meta-association" and "rename meta-attribute" and are related to object-oriented refactoring patterns. These operations, which we will call *delta operations*, are reusable. Conveniently, migration operations can be generated from sequences of delta operations. It is important that any possible evolution can be modelled. There is a general consensus that the proposed sets of delta operations do not suffice. Herrmannsdörfer et al. try to solve this problem by repeatedly extending their list of delta operations [16]. In addition, they support customised evolution. This ensures expressiveness, but the migration transformation must be implemented manually. Interestingly, they classify migration operations with respect to their reuse: operations can be reused across different meta-models, only across models of the same meta-model, or not at all. They observe that such unreusable operations, which they call model-specific coupled changes, are only needed in less than 1% of the migrations in their industrial use cases, when using an elaborated list of operations.

Gruschko et al. [11] distinguish between non-breaking, resolvable and unresolvable delta operations. Non-breaking operations do not render models non-conformant to the new meta-model, and hence do not require co-evolution. For example, the addition of an optional relationship in the meta-model is a non-breaking operation. Because of the optional nature of the change, none of the instance models have to be co-evolved in order to conform to the new meta-model. The instance models simply do not make use of the new language feature. Resolvable operations cause inconsistencies that can be resolved by automated co-evolution. For example, renaming a class in the meta-model requires renaming of the class usages in the instance models. Model co-evolution for unresolvable operations requires additional information in order to execute. For example, when an "add obligatory meta-property"-operation is performed on a meta-model, a new property is created for each instance. However, the initial value of this feature is unknown, as it differs from model instance to model instance. There are two ways to solve this problem. On the one hand, a default value or expression can be given. Although this default value must be manually included in the migration transformation, the instance model can be migrated fully automatically. However, this does restrict the migration to one default scenario, which might not be accurate for each co-evolving artefact. On the other hand, one could manually adapt each model. Though the only correct solution in some cases, it can be a tedious job to perform manual co-evolutions on each model separately.

To illustrate the different kinds of delta operations, the operations proposed by Cicchetti et al. [5] are shown in Table 1. Referring to CRUD operations, they classify changes as additive, subtractive and updative. After additive changes have been applied, more models are valid. In other words, the modelling space of the language is enlarged. Subtractive changes reduce the modelling space, and updative changes ensure the same size. A default migration operation is included for each delta operation. As expected, the non-breaking operations do not need a migration operation, the resolvable operations have a useful migration operation, and the unresolvable operations have a default operation that does not always have the desired migration effect. If necessary, similar operations can be created for the constraint language, as it also has a meta-model [42]. With such operations, evolution of the part of the meta-model that specifies the static semantics can be expressed. Note that we added an inverse operation for every operation as we will need it later on. By definition, performing a delta operation on a meta-model, followed by its inverse operation, results in the original meta-model.

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

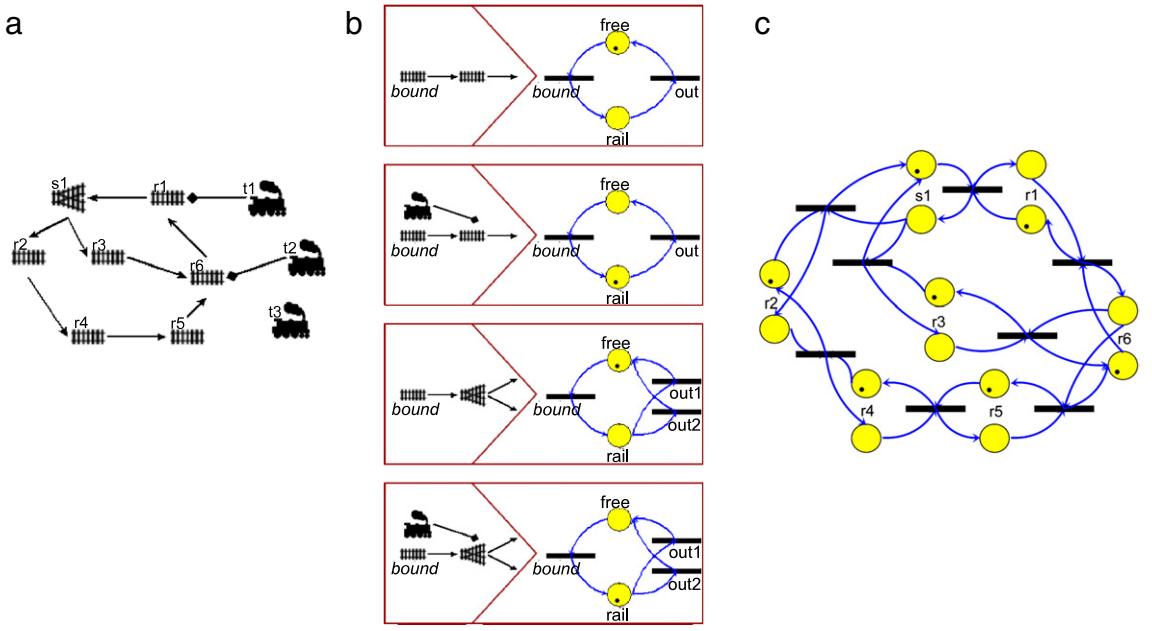


**Fig. 2.** (a) A model in the *TrainSim* DSML; (b) The semantic mapping transformation *[[.]]* onto Petri nets in the form of rules; (c) The "meaning" of the *TrainSim* instance model.

From the above literature study, we create a classification of instance model migration.

- ***Ad hoc transformation vs. coupled transformation.*** As previously mentioned, migration transformations are naturally modelled as model transformations. A migration transformation can be composed in a non-structured, ad hoc manner, or by coupling it to the meta-model changes. For coupled transformation, different kinds of changes require different migration operations, which can be classified as follows:
- ***Non-breaking vs. breaking changes.*** Non-breaking and breaking changes can be distinguished. Breaking changes are often divided into the following two types:
- ***Resolvable vs. unresolvable changes.*** Resolvable changes can be automated. According to many authors, changes can be unresolvable, meaning that the according migration operations must be created manually [5,11,16,50,51].

Orthogonal to this classification, there is the matter of execution of the migration transformation.

- ***Automatic migration execution vs. manual intervention.*** According to some authors, not only the creation of migration operations, but also the execution of the migration transformation can require manual intervention [5,11]. Such a manual intervention can be different for each instance model.

All of these choices are incorporated into our approach.

## 4. Context: Running example

Fig. 2(a) shows a model written in the *TrainSim* DSML which we will use to illustrate our approach to language evolution. Using the *TrainSim* language, the behaviour of trains riding on rail segments can be modelled: *Rail* and *Split* components can be connected into arbitrary networks. *Train*s can be located on *Rail*s and *Split*s, which is visualised by a link between them. The semantics (behaviour in this case) of a *TrainSim* model are given by a semantic mapping onto Petri nets [39]. We use a rule-based (graph) transformation, *TrainSim2PetriNet* to model and execute the mapping. A simplified version is shown in Fig. 2(b). *TrainSim2PetriNet* consists of a number of transformation rules, each containing a Left-Hand Side (LHS) and a Right-Hand Side (RHS) pattern. The execution of an individual rule starts by checking for the occurrence of the LHS pattern in the transformation input model (also known as the "host graph" in graph rewriting). If one or more matches are found, one is selected and the matched pattern in the host graph is replaced by the RHS pattern. If a rule does not match, the next one is tried, until no more rules are applicable. The particular order in which rules are tried is determined by the "rule scheduling" strategy. For our simple transformation, rules are tried in order of priority (highest priority on top and lowest priority at the bottom in Fig. 2(b)). The topmost rule states that each *Rail* that is connected to a *bound* (*i.e.,* a "context" element that matched in a previous step) *Rail* is transformed into two places, *free* and *rail* and an *out* transition, and is attached to the bound transition. The passing of a context (also known as "pivot") between rules is intuitive and may drastically increase performance of the transformation execution. If the *free* place contains a token, the corresponding *Rail* is free. If the *rail* place contains a token, the corresponding *Rail* is busy. As it turns out, by this construction, a *Rail* can never be both free and busy at the same time in its semantic domain (Petri nets) representation. The three other rules are similar, distinguishing between *Rail*s and *Split*s, which have two *out* transitions. When the model of Fig. 2(a) is transformed using *TrainSim2PetriNet*, it results

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

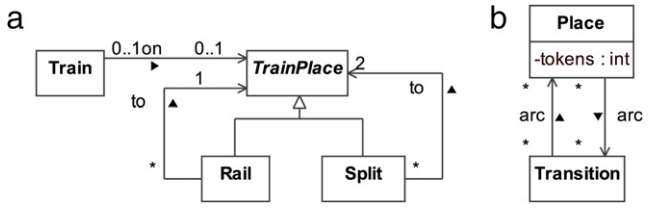


**Fig. 3.** A meta-model for the (a) *TrainSim*, and (b) *PetriNet* domains.

in the Petri net of Fig. 2(c). The behaviour of the model can now be studied using known analysis and simulation techniques for the Petri net formalism. This brings up the issue of mapping back the properties found at the Petri net level to the *TrainSim* domain. Users of a DSML should be shielded from semantic domain intricacies. As an example, the analysis results might report "no, there is no deadlock". The heart of the problem is the need for an explicit representation of properties, at the DSL level, as well as at the level of the semantic domain. Furthermore, it must be possible to trace back these analysis results from the semantic domain to the DSML. For example, when deadlock is detected in the Petri net, it is useful to show the cause of that deadlock in the *TrainSim* model. One way to do this, is to provide a backward transformation from Petri net to *TrainSim* to transform the found Petri net deadlock marking to a *TrainSim* model. Another, more realistic approach is to add traceability information to the forward transformation *TrainSim2PetriNet*. When the transformation is executed, information about the relation between elements of the two formalisms is recorded. This way, the location of analysis results or errors that are found in the semantic domain, can be pinpointed in the DSML. Galvão and Goknil provide a survey of traceability in the context of MDE [8].

A possible meta-model (in the Class Diagram formalism) describing the syntax of the *TrainSim* DSML is given in Fig. 3(a). *Rail*, and *Split* are both subclasses of the abstract *TrainPlace* class, on which at most one *Train* can be located. A *Train* can also be located at no *TrainPlace*. In this context, the direction of a link is modelled by the navigability of the association. From a *Rail*, there must be exactly one outgoing link to another *TrainPlace*, and from a *Split*, there must be exactly two. As a consequence, a *TrainSim* model always forms a closed circuit. The syntax of Petri nets is shown in Fig. 3(b): *Place*s and *Transition*s can be connected by means of directed *arc*s, and *Place*s can hold a number of *token*s.

Throughout this paper, evolutions of the *TrainSim* DSML will be shown. This will serve as a running example of language evolution.

## 5. Evolution of modelling languages

Evolution of modelling languages as described in the related work section implements automation to some extent. Current approaches deal with the co-evolution of instance models. Nevertheless, there are other artefacts that might have to co-evolve. This section presents an exhaustive survey of all possible types of evolution and co-evolution in the MDE context.

### 5.1. Syntactic evolution

To obtain insight into the consequences of evolution, let us go back to Fig. 1 and exhaustively explore all possible evolution scenarios. When $MM_{Lang}$ evolves, all instance models $m$ have to co-evolve, as discussed in Section 3.2. However, as the relations of Fig. 1 suggest, the evolution of $MM_{Lang}$ might affect other artefacts. First, similar to $m$, the domain and/or image of transformations such as $\kappa_i$, $\pi_i$, $T_j$ and $[[.]]$ may no longer conform to the new version of the meta-model. As a consequence, these transformations must co-evolve. This makes all conformance relations valid once again, which means that the system is syntactically *consistent* again. *Consistency* pertains to a collection of models of a system at a *single point in time*. Meta-model evolution requires that model instances and related transformation models co-evolve.

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

Secondly, changes made to one meta-model can have repercussions on another meta-model. For example, when a meta-element is added to a meta-model, a new meta-element is often also added to the meta-model of the concrete syntax(es) in order to be able to visualise this new construct. Thus, there is a need for meta-model co-evolution.

Thirdly, until now, we only discussed meta-model evolution as the driving force behind co-evolution. Evolution of other artefacts, such as instance models and transformation models, should also be taken into account. Normally the case of the evolution of a model $m$ does not enforce a change of a meta-model: related models such as $T_j(m)$ can co-evolve by executing the appropriate transformations $T_j$. Note, however, that a co-evolved model may itself be a meta-model, thereby possibly triggering a cascade of further co-evolutions.

The case of the evolution of a transformation model can become complex. Often, the evolved transformation simply has to be re-executed on each model in this domain. This restricts a transformation evolution to remain compliant with its source and target meta-models, which may not be desired. For example, a new language might be created by mapping rules for

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

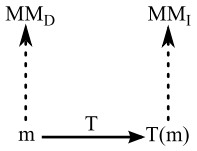


**Fig. 4.** Basic co-evolution schema.

each language construct of an existing language. This is particularly convenient for creating a concrete syntax language. The aforementioned complications stem from two special cases of transformation evolution. Firstly, the evolution of either the parsing or rendering function requires the other one to co-evolve in order to maintain a meaningful relation between abstract and concrete syntax. Such a co-evolution can be generalised to any bidirectional transformation. Secondly, the evolution of the semantic mapping function requires a means to reason about semantics, which are discussed in the next section.

### 5.2. Semantic evolution

As mentioned above, semantics of a model are defined by means of a semantic mapping function. Analyses can be performed on models in this semantic domain (*e.g.,* finding state invariants in a Petri net). The results of these analyses can be considered *properties* of the model. A semantic mapping function is constructed in such a way that some properties hold both for a model and for its image under the semantic mapping. These common properties have to be maintained throughout evolution. An evolution is a semantic evolution if the set of properties that hold changes. This typically happens when the requirements of a system change.

In general, a model $m$ (whose semantics are given by a semantic mapping function $[[.]]$) has properties $P(m)$ that are identical to the properties evaluated in the semantic domain $P([[m]])$. If a model $m$ with properties $P(m)$ in a formalism evolves to $m'$, then $P(m')$ must still be identical to $P([[m]])$. Of course, when semantic changes are implemented on purpose (these are new requirements of the system, formalised by a change in the set of properties) they have to be taken into account when comparing properties before and after migration. In other words, it must be guaranteed that they are equal modulo the intended changes. When two versions of a system are (a) equal modulo their intended syntactic and semantic changes and (b) syntactically consistent, the evolution of the system is said to be *continuous*. *Continuity* pertains to a collection of models of a system as they evolve *over time*. Only continuous evolutions are deemed useful.

### 5.3. Research goal

After sketching the phenomenon of modelling language evolution and introducing new terminology, we can reflect on the research objectives of this paper:

- the *cause* of this research is the observation that modelling languages evolve;
- the *goal* is to maintain consistency and continuity;
- as a consequence, there is a *need* for co-evolution of modelling artefacts;
- the *approach* to achieve co-evolution is to devise *migration* operations.

This paper presents guidelines to achieve these objectives. The aim is to be *complete*, meaning that all possible scenarios are covered. In order to achieve this, we break down the phenomenon of modelling language evolution, so that every single aspect can be explored exhaustively.

## 6. Deconstructing evolution

As shown by the many examples in the previous section, the concept of language evolution is heterogeneous in terms of affected modelling artefacts. In order to deal with this problem uniformly, we first look for basic building blocks for evolution. First, we dissect the consequences of language evolution in a modelled system, and the amalgamation of the basic building blocks is discussed. Then, we dig deeper into the evolution action itself. This leaves us with the proper tools to build the framework in Section 7.

### 6.1. Deconstructing evolution consequences

As discussed in the previous section, there are many possible situations for which co-evolution can occur. These possibilities can be mapped onto the architectural model of a system using the structure of Fig. 1. When we look more closely at the possible scenarios, similarities can be distinguished in the steps that are taken in the co-evolution scenarios. As consistency and continuity are desired during system evolution, co-evolution is in fact the (chained) relationship between meta-model, model and transformation (instead of only meta-model and model, as suggested in related work). Thereby, it is important to note that unidirectional transformations have a domain (*i.e.,* the language that is translated) and an image (*i.e.,* the language this domain is translated to). In other words, for language evolution, both *incoming* and *outgoing* transformations must be dealt with separately. Therefore, we can reduce the problem of co-evolution to the diagram in Fig. 4, where a domain meta-model $MM_D$ and an image meta-model $MM_I$, and their instance models are distinguished, with a

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

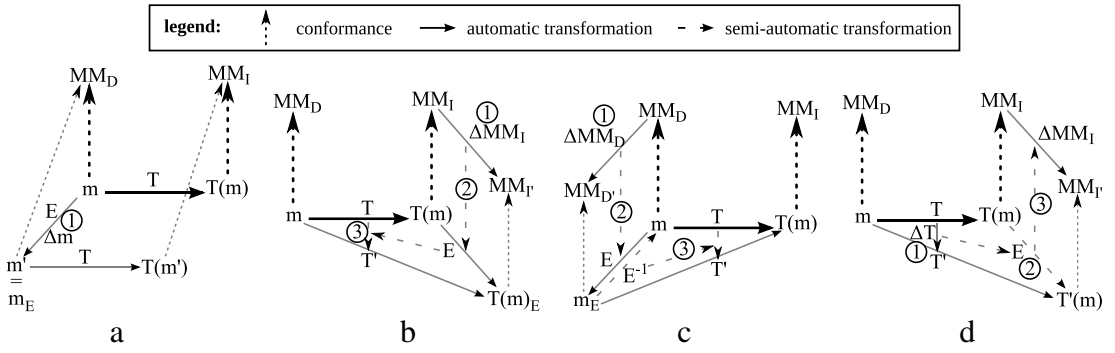


**Fig. 5.** Co-evolution in (a) model evolution, (b) image evolution, (c) domain evolution and (d) transformation evolution.

transformation $T$ in between. Again, arrows denote transformations and dotted arrows denote "conforms to"-relationships. The diagram reduces the problem of evolution to its bare essence.

The possible evolution scenarios can always be broken down into a few basic ones which are depicted in Fig. 5. Small, dashed arrows denote a (semi-)automatic generation. Each diagram starts from the relation between domain and image meta-models, given in Fig. 4. With these four basic scenarios, every possible evolution of Fig. 4 is covered. These are explained in detail in the following paragraphs.

### 6.1.1. Model evolution

Fig. 5(a) shows model evolution. Although in the diagram $m$ evolves, model evolution occurs in fact when either $m$ or $T(m)$ evolves. This kind of evolution is included for completeness, but has quite simple consequences for other artefacts. Some model $m$ evolves to $m'$ (still conformant to $MM_D$). In step 1 (the only step), a delta model $\Delta m$ is constructed (either automatically or manually) that models the evolution of $m$ to $m'$: $m' = m + \Delta m$. The evolution itself is typically represented as a migration transformation $E$, which is the identity transformation for all models in language $D$, with the exception of $m$. If the model is the input of a transformation, as depicted in Fig. 5(a), it is desirable that this transformation is executed again. The transformations whose execution results in the model (in Fig. 5(a) this would mean that $T(m)$ evolves rather than $m$) do not have to be executed again.

In the context of the *TrainSim* example, models could evolve via the insertion, removal or redirection of *TrainPlaces* (which translates to changing the topology of the modelled network) or *Trains* (which translates to changing the configuration of trains within the network). In both cases, $\kappa_i$ and *[[.]]* transformations (as shown in Fig. 1), as well as any other transformation $T_i$ from model $m$, remain valid since no meta-model evolution has occurred. Hence, no further co-evolution is required. Note however, that it might be desirable that some of the transformations are re-executed. For example, if one wants to use the Petri net for analysis, the semantic mapping *[[.]]* must be applied again.

### 6.1.2. Image evolution

Image evolution is shown in Fig. 5(b). Suppose that a meta-model $MM_I$ evolves to $MM_{I'}$. In the context of the *TrainSim* example of Fig. 2, a possible image evolution is the evolution of the target meta-model for the semantic mapping transformation (in this case, Petri nets). For clarity, we use a very simple example of evolution for the Petri net meta-model of Fig. 3(b). Suppose that the name of the meta-attribute *tokens* changes to *numberOfTokens*, to stress that this attribute is just a numeric value. In step 1, a delta model $\Delta MM_I$ is constructed to represent the difference between meta-models $MM_I$ and $MM_{I'}$. In the example, $\Delta MM_I$ can be represented as the operation `RenameMetaElement(Place.tokens, ''numberOfTokens'')`. If a structural representation is preferred, the changed attribute can be tagged as updated (highlighted green for example, in concrete visual syntax).

In step 2, a migration transformation $E$ is generated from $\Delta MM_I$. The execution of $E$ co-evolves models $T(m)$ to models $T(m)_E$ such that they conform to the new meta-model $MM_{I'}$. In the simple example, this could be done using a simple search/replace operation on the representation of each Petri net instance model.

Moreover, the execution of transformation $T$ has to result in valid models (*i.e.,* models that conform to $MM_{I'}$). Consequently, $T$ has to co-evolve to a new transformation $T'$ (step 3), which is able to transform every possible model $m$ conforming to $MM_D$ to an appropriate model $T(m)_E$ conforming to $MM_{I'}$. The diagram presents an intuitive solution for the generation of this $T'$: for every model $m$, $T'(m) = E(T(m))$ holds, and thus we have $T' = E \circ T$. Hence, the co-evolved transformation $T'$ can be obtained simply by composing transformations $T$ and $E$. The trivial way of doing this is to execute $E$ on the output model of $T$. Optimising this composition is a challenge in its own right. In our renaming example, the *TrainSim2PetriNet* transformation can be evolved without digging into the transformation model: each *TrainSim* model can be transformed to the new version of Petri nets by first applying *TrainSim2PetriNet*, resulting in an old version Petri net model, followed by the execution of the migration transformation $E$, which transforms models of the old version to the new version. Resulting models will conform to the evolved Petri net meta-model. Of course, this setting is only valid for simple language evolutions. $T' = E \circ T$ does not hold for the entire evolved image $I'$.

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

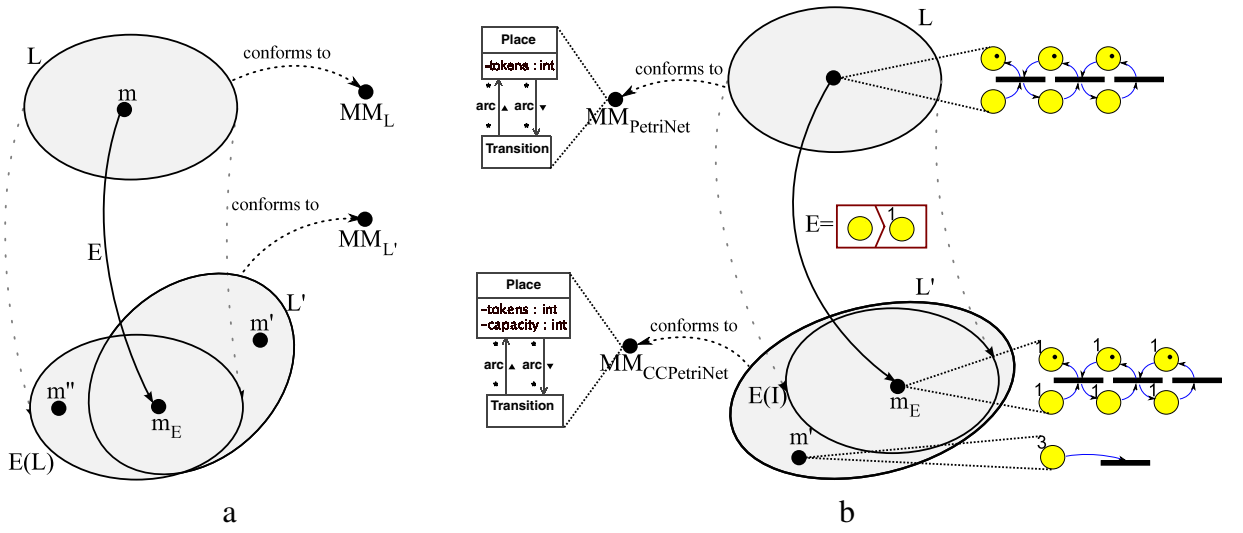


**Fig. 6.** Set representation for evolution with (a) the projection problem after migration and (b) example of the case of $L' \setminus E(L)$.

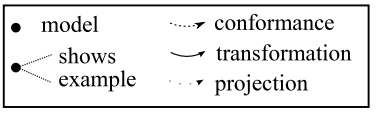


**Fig. 7.** Legend for the set representation figures depicting the projection problem.

To clarify this, a phenomenon we call the *projection problem* must be explained. When a language $L$ evolves to $L'$, $E$ is created to migrate instance models. Ideally, the image of the migration transformation $E$ is equal to the evolved language $L'$. Unfortunately, it is not always the case that the migration transformation projects the original language to the evolved language. Fig. 6(a) shows the projection problem. Ellipses are languages, or sets of models, conforming to the denoted meta-model. Arrows are transformations. Fig. 7 acts as legend for the visual notation used in Figs. 6, 8 and 9. A language $L$ evolves to $L'$, and instance models such as $m$ are migrated by $E$ to $m_E$. There are two different cases where $E(L) \neq L'$, namely $L' \setminus E(L) \neq \emptyset$ (*e.g.,* for $m'$) and $E(L) \setminus L' \neq \emptyset$ (*e.g.,* for $m''$). The case of $L' \setminus E(L)$ occurs for additive changes. Models like $m'$ are valid but cannot be the result of the execution of $E$. In other words, $E$ does not exactly project $L$ to $L'$. The other case of $E(L) \setminus L'$ only occurs if $E$ is not well implemented. Models like $m''$ are models that are the result of the execution of $E$, but are not valid in the new language $L'$. For example, this would happen if no proper $E$ is composed for a subtractive change. This is a matter of faulty implementation instead of an inevitable structural problem. Therefore, we will only take the case of $L' \setminus E(L)$ into account for the projection problem.

Fig. 6(b) shows an example of the projection problem resulting in models that are in $L' \setminus E(L)$. The same layout is used as in Fig. 6(a). The language of Petri nets evolves to capacity constrained Petri nets, by an additive change "add obligatory meta-property". The new property *capacity* denotes the maximum number of tokens a *Place* can hold. In concrete syntax, the maximum capacity is visualised on the upper left side of each *Place*. The migration of instance models $E$ simply introduces a default value of 1 for the new obligatory property, as suggested in Table 1. The model in $L$ is thereby migrated to the model in $E(L)$, and its structure is preserved. However, new models are possible, such as the one with a *Place* with a *capacity* of 3, in $L' \setminus E(L)$. The fact that this model is out of the scope of $E$, will add some restrictions on the use of $E$.

Because of the projection problem, $T' = E \circ T$ does not always hold for image evolution. Following Fig. 4, Fig. 8(a) shows a model $m$ conforming to $MM_D$ that can be transformed by $T$ resulting in the model $T(m)$ conforming to $MM_I$. The image $MM_I$ evolves to $MM_{I'}$ by an additive operation, which results in $I'$ being larger than $E(I)$. Again, the addition of a non-obligatory class can be taken as an example. If $T'$ is acquired by executing $E \circ T$, it turns out that semantic information is not taken into account, as the model is transformed to the language $I$ first, which does not include the newly added semantic information. However, it is generally desirable that the evolved transformation $T'$ can explore the full power of the new version of the image language $I'$.

In Fig. 8(b) an example is shown. The example is based on the evolution from Petri nets to capacity constrained Petri nets of Fig. 6(b). Now, the *TrainSim2PetriNet* transformation is included with *TrainSim* as domain. If it is stated that $T' = E \circ T$, then $T'$ transforms the *TrainSim* model to the model in $E(I)$, by first executing the original $T$, and migrating the model in $I$. However, this way, the changes that were deliberately applied to the Petri net language are ignored. For example, if the mapping to Petri nets would exist in order to analyse for deadlocks, then it would be desirable that after the evolution to capacity constrained Petri nets, the *TrainSim* model would be transformed to the more concise model in $I' \setminus E(I)$, containing only one *Place* with a *capacity* of 3. In that case the exact position of *Trains* is lost, but this information is unimportant for

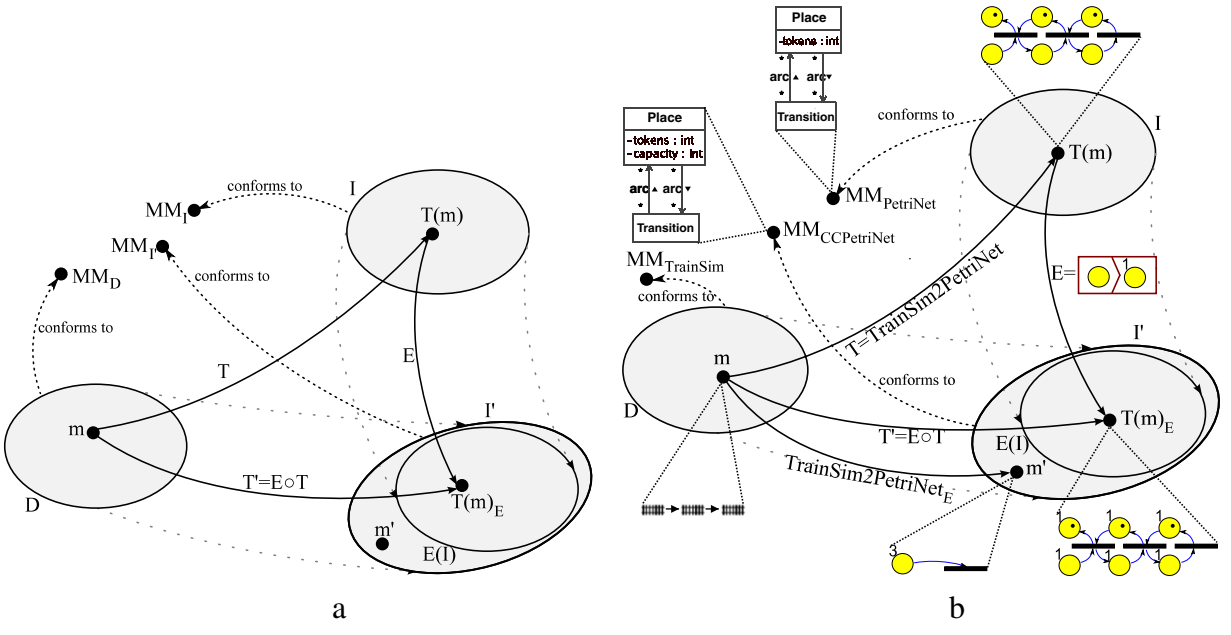


**Fig. 8.** The projection problem in image evolution: (a) $T' = E \circ T$ does not hold and (b) an example.

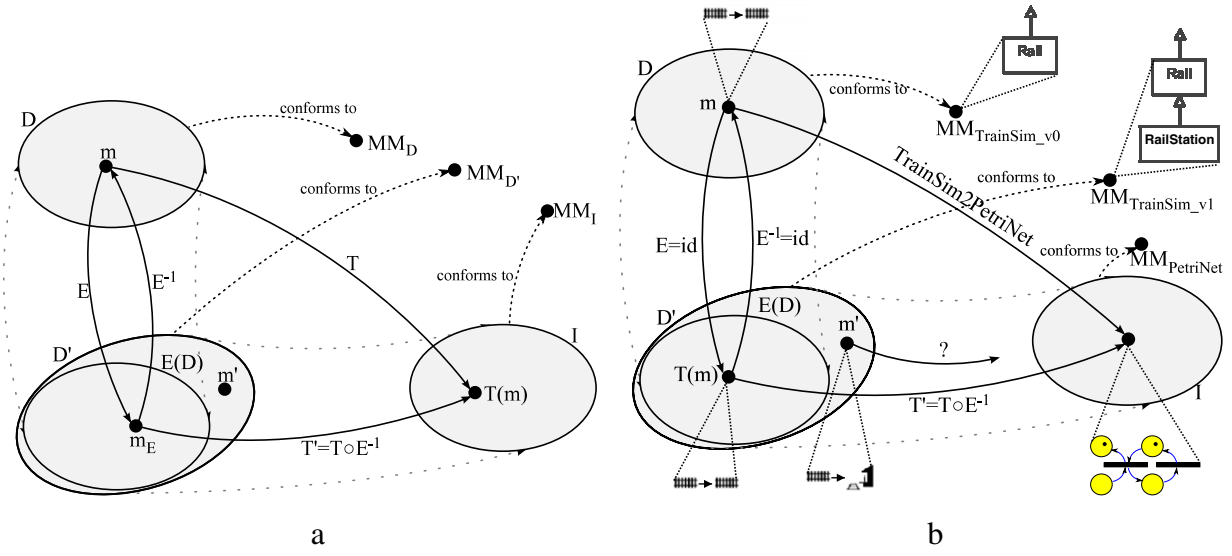


**Fig. 9.** The projection problem in domain evolution: (a) $T' = T \circ E^{-1}$ does not hold and (b) an example.

deadlock analysis. Despite the projection problem, the above principles will prove to be useful in our final approach, as presented in Section 7.

### 6.1.3. Domain evolution

Fig. 5(c) shows domain evolution, where $MM_D$ evolves. The artefacts that co-evolve are similar to those in the image evolution scenario. However, co-evolved transformations $T'$ can now be obtained by $T' = T \circ E^{-1}$ which implies the need to construct an inverse transformation $E^{-1}$. More specifically, in order to transform models conformant to the new version of the language to the image language, they are first co-evolved *back* to the old version, and then the original transformation can be applied. Again, the transformation did not have to be adapted. In the context of the *TrainSim* example of Fig. 2, domain evolution occurs when the *TrainSim* meta-model of Fig. 3(a) changes.

In domain evolution, the projection problem can also occur, as shown in Fig. 9(a). Again, during transformation where $E(D) \neq D'$, critical information about the model can be lost when migrating it back to its old version using $E^{-1}$. Again, this occurs for additive changes. Also, it is desirable that the transformation $T'$ can be applied to its entire domain, including $D' \setminus E(D)$. This part of the transformation cannot be obtained by $T' = T \circ E^{-1}$ when $E(D) \neq D'$. One could intuitively think

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

that subtractive changes also result in a projection problem, as for these changes new information should be "invented" when executing $E^{-1}$. $E^{-1}$ does not have to extend the models again to their original form because the language was reduced deliberately. An example of the projection problem for domain evolution is shown in Fig. 9(b). In the *TrainSim* meta-model as shown in Fig. 3 (visualised partially) a new class *RailStation* is added as a non-breaking, additive change "add non-obligatory meta-class". Since the change is non-breaking, $E$ (and consequently $E^{-1}$) is the identity transformation *id* that applies no changes at all to the model. Similar to image evolution, $T' = T \circ E^{-1}$ applies for models in $E(D)$. However, models in $D' \setminus E(D)$ that use the new language construct (which is visualised by an icon of a station with a waiting person next to a rail) cannot be transformed by $T'$. This is problematic, as it should be possible that every possible *TrainSim* model is transformed to a Petri net.

#### 6.1.4. Transformation evolution

Fig. 5(d) shows transformation evolution. For example, in the context of *TrainSim*, the semantic mapping *TrainSim2PetriNet* could be adapted in such a way that the number of *TrainPlace* visits is recorded in a place. If transformations evolve according to a delta model $\Delta T$, it is possible that they only have to be executed once again. In this case, the changes on the transformation are limited: the image of the evolved transformation $T'$ must conform to $MM_i$ (*i.e.,* the existing Petri net language). As previously discussed, other artefacts such as the image meta-model might also co-evolve. In such cases, a migration transformation $E$ must be constructed from which a delta model $\Delta MM_i$ can be derived. In the example, a timed, capacity constrained or coloured Petri net could be needed as a result of a change in *TrainSim2PetriNet*. Note that such a decision on the co-evolution of the meta-model highly depends on the intention of the evolution and is therefore done manually. The construction of the migration transformation $E$ helps in this process, but can be left implicit at this stage. However, if left implicit, $E$ will be obtained in a later stage when the scenario of meta-model evolution is carried out. Consequently, the issue of the combination of evolution scenarios arises, which is discussed in general in the following section.

### 6.2. Evolution scenario amalgamation

Using a combination of these basic four scenarios, all possible evolutions can be carried out. For unresolvable changes, a general $E$ cannot be found. Note also that the projection problem exists, so automated co-evolution is not always possible. Due to the projection problem, transformations have to support the models in $L' \setminus E(L)$.

In the context of the *TrainSim* example, it is conceivable that a meta-modeller desires domain and image evolutions. For example, that he/she would need to rename *Split* in the *TrainSim* meta-model, and *tokens* in the Petri net meta-model. The co-evolution of the instances of both formalisms is independent of each other. The *TrainSim2PetriNet* transformation, however, co-evolves for both evolutions. After both are applied, the co-evolved transformation will simply be obtained as follows $T'(m) = E_{img}(T(E^{-1}_{dom}(m)))$. In general, evolutions can be chained in transformation co-evolution.

Simply combining basic evolution scenario solutions to solve complex scenarios may yield sub-optimal results. Firstly, domain and image evolutions might happen at the same time because they are closely related. In the *TrainSim* example, the meta-modeller might want to add the notion of capacity to a *Rail*. At the same time, maximum capacities are introduced in the Petri net formalism. In that case, it can be desirable that the *TrainSim2PetriNet* transformation is only migrated once, taking both evolutions into account at the same time. If not, some artificial mappings might be necessary when the *TrainSim2PetriNet* transformation is migrated for only one of the evolutions. Secondly, the performance of combining scenarios might be sub-optimal. In a system under development, an additional transformation must be executed for each additional evolution. Hence, the total number of migration transformations to be executed after an evolution is $\mathcal{O}(n \cdot v)$, with $n$ the number of artefacts that are co-evolved and $v$ the number of existing versions. It may be possible to obtain the same output models while executing less and/or simpler transformation rules. An open research problem is the (semi-)automatic merging of sequences of transformations into more efficient (but equivalent) transformations. Hereby, issues such as order dependence of transformation rules must be overcome. Because they are more modular, at a suitable level of abstraction (hiding matching and rewriting issues), and explicitly meta-modelled (allowing for higher order transformations) it is likely that rule-based transformation specifications will be more amenable to such optimisations than pure code-based ones. Nevertheless, complex rule pattern independence analysis [14] has to be conducted in order to reason about the composability of transformation rules.

### 6.3. Deconstructing evolution actions

In this section, evolution as an operation is deconstructed into manageable, reusable parts. As described in Section 3, the literature classifies model changes as non-breaking, resolvable breaking and unresolvable. Table 1 shows how delta operations can also be classified as updative, additive or subtractive, referring to CRUD (create/read/update/delete) operations.

For each of the delta operations of Table 1, there is a migration operation that can be used to co-evolve instance models. Some migration operations can be implemented as simple scripts, but many more migration operations can be implemented intuitively as rule-based graph transformations. In most transformation tools, such transformations require a well-defined

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

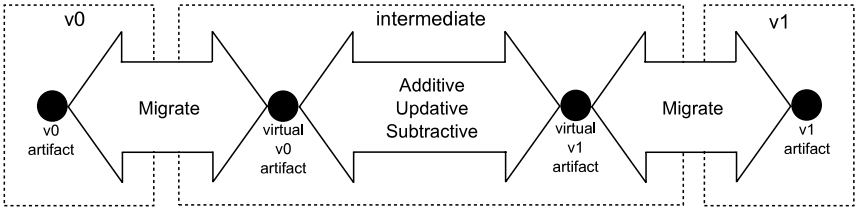


**Fig. 10.** The transformation pipeline.

domain and image meta-model. As we want to keep the migration transformation as separate migration operations for modularity, we will need intermediate meta-models for each migration transformation. Indeed, the result of the first of many migration operations will not yet be a model conforming to the meta-model of the new version of the language, but a meta-model "in between" the two versions.

To solve this problem, we suggest to use a relaxed "intermediate meta-model", which is the result of merging the meta-models of both versions [54]. This intermediate meta-model is generated from the old meta-model, and the delta operations. In case of additive operations, the new elements are added to the intermediate meta-model as non-obligatory language concepts. In case of subtractive operations, the removed elements are made non-obligatory. In case of updative operations, both versions of the element are non-obligatory in the intermediate meta-model. As a result, every model that conforms to the old meta-model, as well as every model that conforms to the new meta-model, can be conceptually expressed in the intermediate meta-model. A step-wise co-evolution from an old version to a new version is now possible. Before and after the execution of each migration step, the model under migration conforms to the intermediate meta-model.

In Fig. 10, the step-wise migration of a version 0 artefact to a version 1 artefact is shown. First, the version 0 artefact is migrated to conform to the intermediate meta-model. This consists of replacing concepts of version 0 with their respective concepts of the intermediate meta-model. The resulting artefact conforms to the intermediate meta-model, but is a "virtual version 0 artefact", as the artefact is semantically identical to its version 0 counterpart. In a sense, the virtual version 0 artefact lies at the edge of the modelling space described by the intermediate meta-model at the side of the version 0 meta-model. Next, the actual migration is performed, by applying additive, subtractive and updative operations. After all these have been applied, the resulting artefact will be a "virtual version 1 artefact". Then, a straightforward conversion to a version 1 artefact is possible. As the bidirectional arrows suggest, it is possible to apply the migration in both directions. This way, the inverse migration transformation that can be used for domain evolution is also captured by Fig. 10.

## 7. A framework for modelling language evolution

In this section, a top-down approach for a framework for evolution of modelling languages is presented. The main requirement for this framework is that it must be complete, which means that it must be able to handle every possible case of language evolution. Furthermore, it must present a solution that is as automated as possible. This means that user intervention must be limited, and directed by the framework. We can distinguish three levels of automation, ordered by the level of user intervention required:

- full automation. This highest form of automation can be implemented in the framework itself. When used, it does not require any manual intervention;
- evolution-specific automation. This form of automation is achieved when it is implemented at the level of evolution itself. As a result, the corresponding co-evolutions can be performed automatically;
- no automation. When manual intervention is needed at the level of the co-evolving instances, there is no automation.

Although full automation is desirable, the other classes of automation are sometimes necessary.

### 7.1. Re-constructing evolution

In Section 6, evolution was deconstructed into manageable basic scenarios. The modelled system of Fig. 1 was broken down into the basic schema of Fig. 4. In this section, we use this breakdown to re-construct all possible evolution scenarios. Then, similarities and differences between scenarios can be investigated and a suitable solution can be customised to maximise automation.

In the basic co-evolution schema of Fig. 4, each of the involved artefacts can evolve, causing some other artefacts to co-evolve. Hence, we assume that each system can be mapped onto (multiple instances of) this schema, which originated from the modelling artefacts shown in Fig. 1. An instance of the basic schema can be identified in a system by identifying a transformation as an instance of $T$. Then, incoming models are instances of $m$ and their meta-model is an instance of $MM_D$. The resulting models are instances of $T(m)$ and their meta-model is an instance of $MM_I$. However, in theory, a transformation has one input model and one output model, this input or output model can be a modular composition of two models (possibly conforming to two different meta-models). An example of a transformation with a composed input model is model merging, and an example of a transformation with a composed output model is generating a trace model together with the output model. In many existing transformation tools, the transformations have combinations of more than one meta-model

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

**Table 2**
Evolution scenarios for the basic co-evolution schema.

| | Changing artefact | | | | |
|---|---|---|---|---|---|
| Type of $T$ | $MM_D$ | $MM_I$ | $T$ | $m_D$ | $m_I$ |
| Unidirectional exogenous transformation | Co-evolve $m_D$ and $T$, execute $T$ | Co-evolve $m_I$ and $T$ | Change $MM_D$ and/or $MM_I$ | Change $MM_D$ | Change $MM_I$ |
| Bidirectional exogenous transformation | Co-evolve $m_D$ and $T$ and $T^{-1}$, execute $T$ | Co-evolve $m_I$ and $T$ and $T^{-1}$, execute $T^{-1}$ | Change $MM_D$ and/or $MM_I$ | Change $MM_D$ | Change $MM_I$ |
| Unidirectional endogenous transformation | Co-evolve $m_D/m_I$ and $T$, execute $T$ | Co-evolve $m_D/m_I$ and $T$ | Change $MM_D/MM_I$ | Change $MM_D/MM_I$ | Change $MM_D/MM_I$ |
| Bidirectional endogenous transformation | Co-evolve $m_D/m_I$ and $T$ and $T^{-1}$, execute $T$ | Co-evolve $m_D/m_I$ and $T$ and $T^{-1}$, execute $T^{-1}$ | Change $MM_D/MM_I$ | Change $MM_D/MM_I$ | Change $MM_D/MM_I$ |
| Semantic mapping function $[[\cdot]]$ | Co-evolve $m_D$ and $[[\cdot]]$, execute $[[\cdot]]$ | Co-evolve $m_I$ and $[[\cdot]]$ | Change $MM_D$ and/or $MM_I$ | Change $MM_D$ | Change $MM_I$ |
| Concrete mapping function $\kappa$ and $\pi$ (rendering and parsing) | Co-evolve $MM_I$, $m_D$ and $\kappa$ and $\pi$, execute $\kappa$ or $\pi$ depending on changing artefact | Co-evolve $MM_D$, $m_I$ and $\kappa$ and $\pi$, execute $\kappa$ or $\pi$ depending on changing artefact | Change $MM_D$ (and consequently $MM_I$) | Change $MM_D$ (and consequently $MM_I$) | Change $MM_D$ and (and consequently $MM_I$) |

describing input or output. In those cases, it is desirable to maintain the modularity of the different formalisms that are used. Therefore, each input or output meta-model is mapped onto a different instance of the basic schema. For example, a transformation that transforms an input model to an output model and a trace model (in order to maintain an explicit relation between the input and output model) has two mappings to the basic schema: one with the meta-model of the output model as $MM_I$, and one with the meta-model of the trace model as $MM_I$.

Additionally, we must take into account the type of transformation $T$. Different kinds of transformations have different co-evolution consequences. Transformations can be (orthogonally) endogenous/exogenous, unidirectionally/bidirectionally, in-place/out-place (we assume preservation of the model after transformation execution, thus out-place transformations) [29]. Endogenous transformations use the same meta-model for its input and output models, while exogenous use different meta-models. Unidirectional transformations are transformations from one model to another, in that specified direction. Bidirectional transformations are represented as a relation between models, which is applicable in both directions. Out-place transformations create a new output model from scratch, while in-place transformations perform changes on the input model itself. Transformations can be used as parsing or rendering functions, semantic mapping or model to model transformation [22]. They can also keep generic links between concepts in different models, which results in traceability between concepts of the model. Although this can greatly facilitate the automation of co-evolution at the implementation side, this is irrelevant from the architectural point of view presented in this section. Also note that transformation models can range from simple search–replace scripts, to XSLT transformations, to rule-based graph transformations. Although the above distinction suffices to express the different kinds of transformations, the concrete mapping functions $\kappa$ and $\pi$ can be considered a special case because of the tight coupling between the meta-models of abstract and concrete syntax. This relation is homomorphic in nature. In order to maintain that homomorphic relation, and a meta-model evolution (*e.g.,* of the abstract syntax) that changes the meta-model structurally requires migration of the other meta-model of the relation (*e.g.,* of the concrete syntax). In Table 2, the possible co-evolution scenarios are explored, where rows represent the kind of transformation, columns represent the artefact that changes. As artefacts of Fig. 4 are used, we assume that every language evolution in a system can be mapped onto one or more cells in this table.

Some conclusions can be drawn from observing the table. The default scenario is the evolution of $MM_D$ or $MM_I$, *i.e.,* the first two columns. Evolutions of $T$, $m_D$ or $m_I$ usually do not cause a change on the meta-level (*i.e.,* the level of the language). Indeed, evolution causes at most execution of $T$, in order to keep the models in the system consistent. This is in particular true, and usually must happen instantly, for multi-view systems [12]. As this does not change anything at the meta-level, it is not taken into account. In some cases, however, a change at the meta-level is desirable. Suppose that a modeller can adapt a language by changing the model. This is useful because the modeller can perform simple modifications to a language without extensive knowledge of language engineering, such as meta-modelling or transformation modelling (in case of adaptation of $T$ itself, the user must understand it of course). Instance-based modification of a language is possible if the modelling environment allows *free-hand* editing (as opposed to syntax-directed editing) [31]. For such a scenario, the last three columns in Table 2 are filled in, as we are interested in language changes.

A second conclusion is that, apart from the concrete mapping, all types of $T$ are very similar. Evolution of a language always leads to co-evolution of instances and transformation models, with a possible execution of $T$ as a consequence. Note that we limit each scenario to the schema of Fig. 4, so, some real-life scenarios might trigger more than one evolution scenario

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

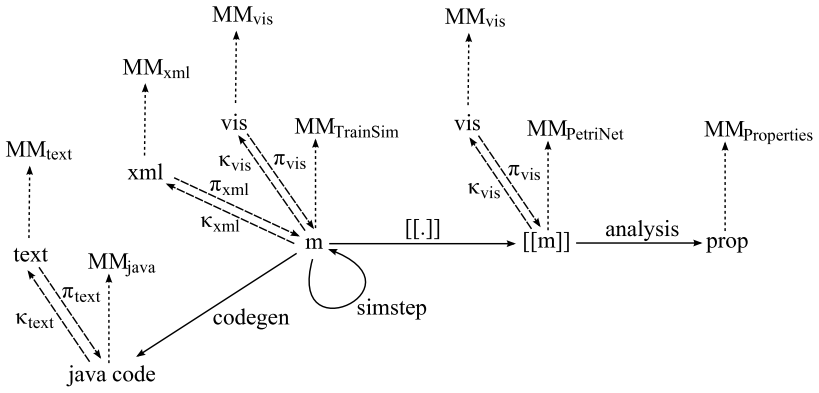


**Fig. 11.** A full *TrainSim* example, including concrete syntaxes, semantic mapping, code generation, simulation and analysis.

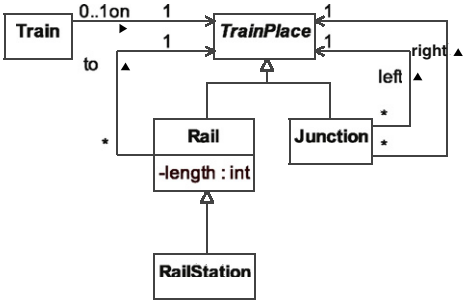


**Fig. 12.** An evolved meta-model for the *TrainSim* domain.

of Table 2. For example, when evolving a meta-model, each of the related transformation models is co-evolved, which can each be mapped onto a table cell. Furthermore, because of the co-evolution consequences, a chain of transformation executions can be triggered.

The concrete mapping is, however, notably different. The abstract syntax must be kept consistent with the concrete syntax(es). The concrete mapping functions $\kappa$ and $\pi$ (for rendering and parsing) can for this purpose be considered a special kind of transformation. Some tools such as Fujaba [9] force the user to explicitly implement a concrete mapping. Abstract syntax concepts can be instantiated using Object Diagrams. Tools such as AToM$^3$ [7] include concrete syntax in the language definition (*i.e.,* attached to meta-model elements). This illustrates the tight coupling as well as the simple one-to-one nature of the concrete syntax mapping.

### 7.2. Running example

Fig. 11 shows the relationship between various language artefacts. The core of the diagram is the *TrainSim* language, modelled by $MM_{TrainSim}$, its concrete syntaxes $MM_{vis}$ and $MM_{xml}$, and its related transformation models *[[.]]* (the semantic mapping, which we already used in the previous example), *codegen* (an operational mapping to Java code), and *simstep* (to simulate trains riding over the track). Note that concrete syntaxes of a formalism, because of their tight coupling with the abstract syntax, are visualised "behind" their abstract syntax. Concrete mapping transformations are visualised as dashed arrows. From *TrainSim* models, which can be represented as XML (for storage), textually (human-readable) and visually, simulation can be done, code can be generated for deployment and execution, and Petri nets can be generated for deadlock analysis. The results of this analysis are represented in the *Properties* language (*True* when there is a deadlock or *False* when there is none).

Reconsider the meta-model for the *TrainSim* domain of Fig. 3. During the development cycle, some changes are made at the meta-level. In the new version of the meta-model, which is shown in Fig. 12, a *Split* is again renamed to *Junction*, and a *Train must* be on a *TrainPlace*. Also, instead of having a *to* connection with multiplicity 2 from the *Junction*, there are two different associations called *left* and *right*, in order to add a notion of direction. Finally, a *RailStation* is also added.

### 7.3. Framework features

Fig. 13 shows a feature diagram [18] of the framework that is explained in this section. Feature diagrams are a model for product families, and they model how different products can be composed. In our context, we use it to describe a *family* of migrations, *i.e.,* the diagram models all possibilities for the evolution of modelling languages. This section offers guidelines to systematically tackle the problems that emerge when modelling languages evolve. The feature diagram shows what

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

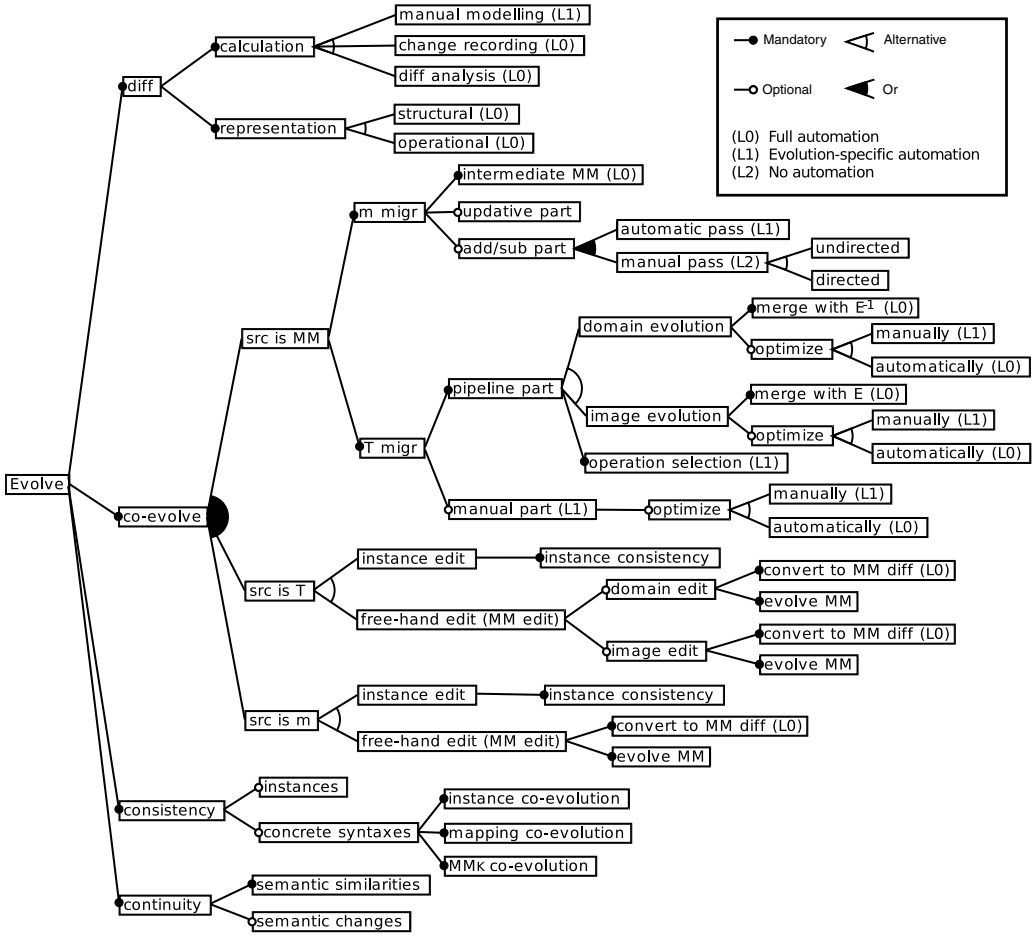


**Fig. 13.** Feature diagram of the framework for evolution of a modelling language.

possibilities have to be taken into account for the creation of a framework. Afterwards, an algorithm is provided. Throughout the remainder of this section, the feature diagram is carefully explained, and features from the feature diagram are referred to in italic font. In order to support evolution (*Evolve* in the feature diagram), the framework must support:

- model differencing (*diff*);
- co-evolution (*co-evolve*) of instance models or related transformation models. Depending on the context, this can originate from meta-model changes (*src is MM*) or changes of another artefact (*src is T* and *src is m*). More than one possibility emerges here: a changing artefact can be an instance model in one mapping to the basic schema, a meta-model causing domain evolution in another mapping to the basic schema, a meta-model causing image evolution in yet another mapping to the basic schema;
- *consistency* of the system;
- *continuity* of the system;

These features are explained further in this section.

### 7.3.1. Difference between models

The evolution of an artefact must be modelled explicitly, in order to derive co-evolution (semi-)automatically. Therefore, the difference between two versions of a model needs to be represented (*representation*). As discussed in Section 3.1, there are two kinds of representations: *operational*, where difference is modelled as a series of edit operations, and *structural*, where difference is modelled as a language element (represented explicitly in a meta-model).

Apart from representation, the *calculation* of the difference model is an essential part of the framework. The difference models can be modelled manually (*manual modelling*), resulting in evolution-specific automation. Full automation can be achieved when the difference is calculated from the two versions of the meta-model (*diff analysis*), or when the changes are recorded at the moment they are applied (*change recording*). Change recording is very accurate, and does not require a complex algorithm to be implemented. However, it is dependent on that particular framework and difficult to visualise. The differences in change recording are conveniently represented as an operational model.

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

**Table 3**
The operational difference model of the evolution.

| nr. | Operation | Type |
|---|---|---|
| 1 | RenameMetaElement(Split, "Junction") | Updative |
| 2 | RestrictMetaProperty(TrainOnTrainPlace, 1, 1, 0, 1) | Subtractive |
| 3 | EliminateMetaProperty(Junction.to) | Subtractive |
| 4 | AddNonObligatoryMetaProperty(Junction, TrainPlace, "left", 1, 1, 0, -1, False) | Additive |
| 5 | AddNonObligatoryMetaProperty(Junction, TrainPlace, "right", 1, 1, 0, -1, False) | Additive |
| 6 | AddObligatoryMetaProperty(Rail, "length", Integer, 1, 1, 1) | Additive |
| 7 | AddNonObligatoryMetaClass("RailStation", [Rail], False) | Additive |

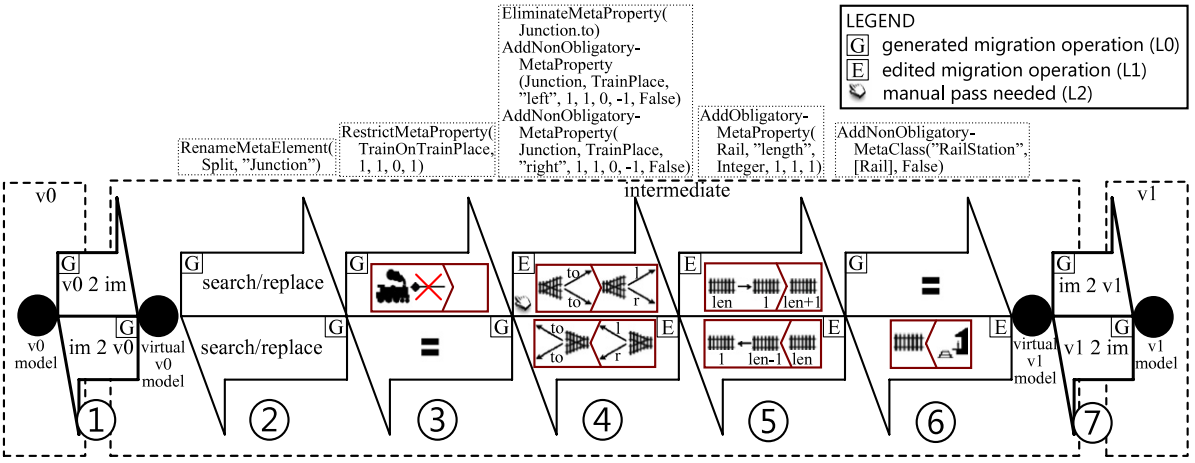


**Fig. 14.** The migration pipeline for the *TrainSim* evolution.

In our evolution from the meta-model of Fig. 3 to the one of Fig. 12, we use an operational representation of the difference model in Table 3. The difference model uses the operations from Table 1. Note that the replacement of the *to*-association is modelled as action 3-5. In this example, the difference model is created manually, but existing automated approaches can be used instead (see Section 3.1).

### 7.3.2. Evolution of meta-models and co-evolution of instance models

When following Fig. 10, a migration model (*m migr*) involves creating an intermediate meta-model, an updative part and an additive/subtractive part. As stated in Section 6.3, the intermediate meta-model (*intermediate MM*) describes all concepts of both meta-model versions, and can be generated fully automatically. The *updative part*, consisting of all updative changes, can be automatically derived from the difference model. As updative changes are not subject to the projection problem, migration operations and their inverse can be generated automatically. The additive/subtractive part (*add/sub part*), consisting of all additive and subtractive changes, might need manual adaptation, so possibly no automation can be applied to some part of the co-evolution. As a result, co-evolution is performed using *automatic pass*(es) and *manual pass*(es). The manual pass can require *undirected* ad hoc adaptation, or can be *directed* by the framework, which points out where manual intervention must be done in the co-evolving artefact.

A migration pipeline is composed for every evolution. When following the example evolution, a pipe as in Fig. 14 can be created. Each big arrow contains the migration operation in each direction for one or more delta operations that are shown in a dotted rectangle above the arrow. The migration operation to migrate from the old version to the new version is shown in the upper half, the one to migrate back from the new version to the old version is shown in the bottom half (and can be ignored for the co-evolution of instance models). A first draft can be generated automatically from Table 3, creating a slot with the default migration operation from Table 1 for each change. For example, the third slot contains a generated migration operation indicating that trains that are not connected with some *TrainPlace*, are removed. In two cases, a slot contains a large equality sign. This means that no migration must be performed, as the respective delta operation is a non-breaking operation (see Table 1). Generated migration operations are marked with a "G" (for "generated"). In a next step, which is done manually (but offers evolution-specific automation), slots can be replaced and combined. When the migration operation of a slot is overridden, it is marked with "E" (for "edited"). In this example changes 3 through 5 from Table 3, as shown in the large dashed rectangle, are combined into the fourth slot of Fig. 14. Occurrences of the old *to* connection are replaced with the *left*- and *right* connections. This migration operation requires a manual pass (indicated by the hand icon that denotes the need for manual editing), where a distinction is made between the "left" and "right" *TrainPlace*. Alternatively, this could have been done randomised and automatically, but critical information might be wrong. Also, the default migration operation

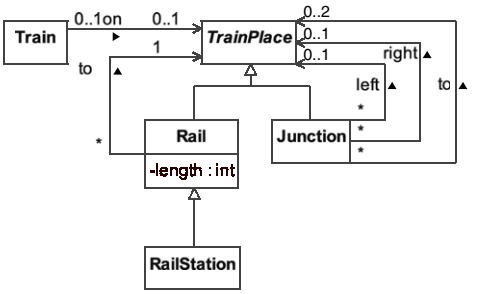


**Fig. 15.** The intermediate meta-model for the *TrainSim* domain.

for change 6, where a length attribute is added, is replaced with a fold/unfold operation (see the fifth slot in Fig. 14). The fold operation will detect a chain of *n Rail*s by greedy matching and will transform it into one *Rail* with length *n*. In order to co-evolve instance models, the model is passed through the pipeline from left to right, co-evolving it in a step-wise manner, using the upper migration operations. This sequence of all migration operations can be called as a whole, the migration transformation $E$ of the evolution. At slot 4, user interaction is required. The co-evolution of all available instance models can be scheduled conveniently by migrating them all to slot 4, then performing the manual pass on each of them, and finally migrating them from slots 5 to 7. The inverse migration operations (the bottom half of the slots) are never used when co-evolving instance models. As discussed previously, the pipeline requires the use of an intermediate meta-model for the *TrainSim* domain, shown in Fig. 15.

Note that this example is deliberately chosen to show the different aspects of our approach. When using migration operations in practice, however, Herrmannsdörfer et al. [16] have shown that in industrial case studies, generated migration operations can be used for more than 99% of the total number of operations. In contrast to the example of Fig. 14, editing migration operations is needed for only a limited number of cases.

### 7.3.3. Evolution of meta-models and co-evolution of related transformation models

In addition to instance models, related transformation models must be co-evolved. A transformation may often be represented as a graph transformation, consisting of graph rewriting rules. However, its implementation can be simpler in some cases, such as an XSLT transformation or a simple script. Transformations can be total, *i.e.,* the whole domain can be mapped onto the image. In many cases, however, some language constructs of the domain do not have an image, so the transformation is partial. In this case, information is lost.

Although a less popular research topic than model co-evolution, the problem was already described by Zhang et al. in 2004 [56]. Levendovszky et al. state that transformation co-evolution is more complex, thus less automatable, than instance model co-evolution [25]. Consider for example the "Add (non-obligatory) meta-class" operation of Table 1. Although not a problem for instance models, it is very well possible that a transformation model must be changed in order to map the new class onto a construct of the image language. Levendovszky et al. call such additive operations "fully semantic transformation operations", but do not provide support for them.

In our framework we divide transformation co-evolution (*T migr* in Fig. 13) into two parts: a pipeline part and a manual part. The first part (*pipeline part*) can be automatically executed using a combination of the pipeline used for instance model co-evolution, and the principles of *image evolution* and *domain evolution* of Section 6.1. This means that the transformation is merged with $E$ (*merge with* $E$) or $E^{-1}$ (*merge with* $E^{-1}$). This process can be optimised (*optimise*) *manually* or *automatically*. As noted, the simplistic approach of image and domain evolution is not directly applicable to additive or subtractive changes. Suppose we apply it directly to the evolution example. The co-evolution of *codegen* in Fig. 11 is a case of domain evolution. The co-evolved transformation would be able to transform *TrainSim* models of version 1: $T'(m) = T(E^{-1}(m))$. However, the information about direction after a *Junction* would be lost after applying $E^{-1}$. This is not desirable, as this information was deliberately added to the new version of *TrainSim*.

The use of the intermediate meta-model plays a key role in the solution to this problem. As concepts of both versions of the meta-model can be modelled in the intermediate model, no information needs to be lost. Therefore, instead of executing the transformation on version 0 instance models, we migrate the transformation so that it is applicable to intermediate models. Following the pipeline of the evolution example of Fig. 14, this means that we migrate the transformation model to a virtual version 0 artefact.

In the case of domain evolution for the semantic mapping *TrainSim2PetriNet*, the mapping is migrated to the intermediate meta-model. Then, when a version 1 *TrainSim* model must be transformed, it must be migrated "towards" the migrated *TrainSim2PetriNet* transformation by transforming it to a virtual version 1 artefact. Theoretically, the migrated *TrainSim2PetriNet* can transform the migrated *TrainSim* model. However, as the transformation is virtually v0, it is possible that the mapping is not total anymore. For example, *RailStation*s, and *left*- and *right* connections will not be transformed. Therefore, the virtual version 1 instance model will pass through some chosen slots in the migration pipeline. In this example, it is desirable that *RailStation*s are converted back to *Rail*s, and *left*- and *right* connections are converted back to one *to*-connection. In other words, the inverse migration operations of only change 7 and changes 3 to 5 of Table 3 are applied

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

on the instance model. As a result, the migrated transformation *TrainSim2PetriNet* is now total again. Note that folded *Rail* sequences are not unfolded, as we use the Petri net models for deadlock analysis, and this has no influence on this analysis. In case of image evolution (as is the case, along with domain evolution, for the *simstep* transformation), the transformation is migrated similarly, and (a selection of) the non-inverse migration operations are applied to the resulting model afterwards, resulting in a version 1 instance model. In the endogenous *simstep* transformation, both image and domain evolves, and it turns out that the desired result is achieved when no additive or subtractive migration operations are performed at all. The virtual version 0 *simstep* transformation can be applied to virtual version 1 instance models.

The first part of transformation migration being automatic (apart from choosing the migration operations that must be used), migration of a transformation model possibly requires a manual part (*manual part* in Fig. 13). Consider the *codegen* operational semantic mapping of Fig. 11. In the evolution example, this transformation is subject to domain evolution. The transformation is migrated to the intermediate meta-model. When executed on a *TrainSim* version 1 instance model, the model is migrated to the intermediate meta-model, and again, *RailStation*s are transformed to *Rail*s. Also, *Rail*s with a length higher than 1 are unfolded to a sequence of *Rail*s with length 1. The *left*- and *right* connections are kept, however, as we want to incorporate this new information in the generated code. Of course, this cannot be done automatically. To do this, the virtual version 0 transformation model is adapted manually to support this new concept by changing the rewrite rule that generates code for splits. Because the migrated transformation model uses the intermediate meta-model, it has access to introduced language concepts such as the *left*- and *right* connections. In conclusion, the framework reduces the manual intervention to a strict minimum, *i.e.,* the addition of new semantics. If an evolution comprises a lot of semantic changes, the amount of effort that has to be put in this manual part will be greater.

Note that, when a series of evolutions are performed, the intermediate meta-model might describe a larger modelling space, as concepts of all evolutions are incorporated. On top of this, the number of slots in the pipeline will grow, as each evolution change is described by these migration operations. Therefore, it is desirable to "freeze" the evolved system at some point, and *optimise* the migrated transformations. This can be done either *manually* or *automatically*. From then on, the system can again be considered version 0.

### 7.3.4. Evolution of other artefacts

If an instance model changes (*src is m*) or a transformation model changes (*src is T*), there are two possibilities: either there is no impact on a meta-model (*instance edit*) or a meta-model changes (*free-hand edit*). For the former, some transformations may have to be executed again in order to maintain consistency among the model instances (instance consistency). The latter is more complex and occurs when a tool supports free-hand editing [31]. In free-hand editing, the modeller can immediately edit an instance or concrete syntax in order to change a language. The conversion of the $m/T$ difference to a meta-model difference (*convert to MM diff*) is supported by the free-hand editor. Whenever the editor applies a change to an instance model in the language, the change is generalised and applied to a (possibly implicit) meta-model. Note that this is typically not done automatically, but rather, suggestions are given. Ultimately, the modeller is presented a meta-model editor to allow manual meta-model changing. This can only be done if the other model is closely related to the original model, by a homomorphic mapping. Homomorphic relations are in this context meta-model/instance or abstract syntax/concrete syntax relations. When the meta-model changes (*evolve MM*), a new evolution is triggered. Note that for a transformation model change, the domain meta-model might be subject to change (*domain edit*) as well as the image meta-model (*image edit*).

### 7.3.5. Consistency and continuity

When a language evolves and instance models and transformation models are co-evolved, the system might have become inconsistent (*consistency*). In many cases, it is desirable to solve this by executing respective transformations so that the model instances are consistent again (*instances*). Note that this can result in a chain of transformation executions. In that case, the order in which different transformations are executed might be relevant for optimisation reasons. After the necessary executions have been done, the system is consistent again. We assume that all evolutions and co-evolutions are done correctly, *i.e.,* they can result in a consistent system.

In addition to the consistency of instance models, we also have to take consistency between abstract syntax and concrete syntaxes into account (*concrete syntaxes*). Due to the homomorphic nature of the concrete syntax mapping and the tight coupling, this problem is usually simpler than other co-evolution problems, as the concrete syntax mapping mainly consists of only simple one-to-one transformation rules. When changing abstract syntax, the meta-models of the corresponding concrete syntaxes must co-evolve. This requires migration of the meta-model of each concrete syntax ($MM_\kappa$ *co-evolution*). Also, $\kappa$ and $\pi$ themselves have to co-evolve (*mapping co-evolution*), and the concrete syntax instances can be migrated as regular instances of a meta-model (*instance co-evolution*). The other way around, when a concrete syntax structurally evolves in a free-hand modelling tool, it requires co-evolution of its abstract syntax. This will in turn trigger co-evolution of the other concrete syntaxes, if available. Apart from the co-evolution on the meta-level, co-evolution also happens for instance models and the transformation model (*i.e.,* concrete syntax mapping).

In order to perform a meaningful evolution, it must be *continuous*. This means that the system is consistent and semantically equal to its previous versions (*semantic similarities*), with intended changes taken into account (*semantic changes*). This can in principle be checked by executing the semantic mapping and comparing the properties on these models by analysing their results. In the *TrainSim* example for instance, a property could be the answer to the question "can two

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

trains block each other in this train circuit?". In order to get a result for a model, the model is transformed to the Petri net formalism (if not already done) and deadlock analysis is performed on the Petri net model. The result is propagated to the *TrainSim* model. After evolution, the same result for this property should be obtained for each model, modulo intended semantic changes.

### 7.4. Algorithm

In this section, we describe an algorithm as the backbone of the proposed framework. In Fig. 16 a class diagram that is used by the algorithm is shown as an illustration. As our domain is MDE, it is not surprising that the root class is *Model*. Its subclasses are:

- *TransformationModel* that can transform models to other models. A *TransformationModel* can be a *ConcreteSyntaxMapping* (which can be a *RenderingMapping* or a *ParsingMapping*), a *MigrationOperation* or a *MigrationPipeline* which is an ordered set of *MigrationOperations*;
- *InstanceModel* that conforms to a meta-model;
- *MetaModel* that must be either an *AbstractSyntax* or a *ConcreteSyntax*;
- *DifferenceModel* that is a model that describes the difference between two models. A *DifferenceModel* is either an *OperationalDiffModel* or a *StructuralDiffModel*.

Further in this section, an algorithm is shown for the *evolve* and *evolveTo* methods. These methods offer a complete framework for language evolution in MDE, that can be used to tackle all types of evolution. In the algorithms, methods that are shown in Fig. 16 are used. Some of the used methods are simple *getters*, more complex methods were discussed previously in the paper. The methods can be implemented as one wishes, so an incremental implementation of the framework is possible using the algorithm, where the implemented methods initially require a lot of manual work (e.g., the manual modelling of the delta model), but gradually is automated (e.g., by "plugging in" the DSMDiff algorithm [27] for calculating the delta model). The used methods are:

- *Model::execute*: executes a transformation with the model as input. The target is known by the transformation;
- *Model::difference*: calculates the difference of two models and returns a difference model (see Section 3.1);
- *TransformationModel::merge*: merges two transformation models. This can be done trivially by appending them (see Section 6.2);
- *TransformationModel::domain*: returns the meta-model of the domain of the transformation;
- *TransformationModel::image*: returns the meta-model of the image of the transformation;
- *MigrationPipeline::steps*: returns the sequence of migration steps that form the migration pipeline. The migration steps from the old version to the intermediate version, and from the intermediate version to the new version, are included (see Section 7.3.2);
- *DifferenceModel::generateDefaultMigrationPipeline*: generates the default migration operations following Table 1;
- *DifferenceModel::convertTo*: converts the difference model of an original model to another model (see Section 7.3.4);
- *DifferenceModel::apply*: applies the difference model to a model, so that the evolved model is obtained;
- *MetaModel::instances*: returns all available models that conform to this meta-model;
- *ConcreteSyntax::parsingMapping*: returns the parsing transformation;
- *ConcreteSyntax::abstractSyntax*: returns the corresponding abstract syntax;
- *AbstractSyntax::generateIntermediateMetaModel*: generates the intermediate meta-model from two meta-models (see Section 6.3);
- *AbstractSyntax::dependentTransformations*: returns all transformations that use this meta-model (for its input language and/or output language). It is assumed that these transformations only have one input language and one output language. If in reality a transformation has two output languages, than this transformation is returned twice, once for the first output language, and once for the second one. This assumption is introduced to improve the simplicity of the algorithms further in this section;
- *AbstractSyntax::concreteSyntaxes*: returns the corresponding concrete syntaxes;
- *AbstractSyntax::renderingMappings*: returns the rendering transformation;
- *InstanceModel::metamodel*: returns the meta-model of this model.

Fig. 16 shows the method in pseudocode for the default case of abstract syntax meta-model evolution. It executes the consequences of a meta-model, according to the new version of this meta-model and the difference model. First, the intermediate meta-model and the migration pipeline are created (lines 2–4). The *INPUT()* method denotes manual intervention. Then, instance models can be migrated (lines 5–9) by iterating over the migration operations and the instance models. Next, the relevant transformations are migrated (lines 10–21) by first customising the pipeline and, if necessary, the transformation itself (lines 11–12). Then, a distinction is made between domain and image evolution (lines 13–20), and the transformation and pipeline are merged accordingly. Note that the pipeline is traversed in the opposite way for domain evolution, so that the inverse migration steps are used. In case of domain evolution, the transformation has to be executed again on all instance models (lines 17–19). Finally, the concrete syntaxes are migrated (lines 22–35). Hereby, the homomorphic relationship between abstract syntax and concrete syntax is used extensively. The concrete syntax meta-model has to be migrated (line 23), as well as the rendering mapping (lines 24–29), the parsing mapping (lines 30–31) and the concrete syntax instances, which can be migrated by executing the migrated rendering mapping (lines 32–34).

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

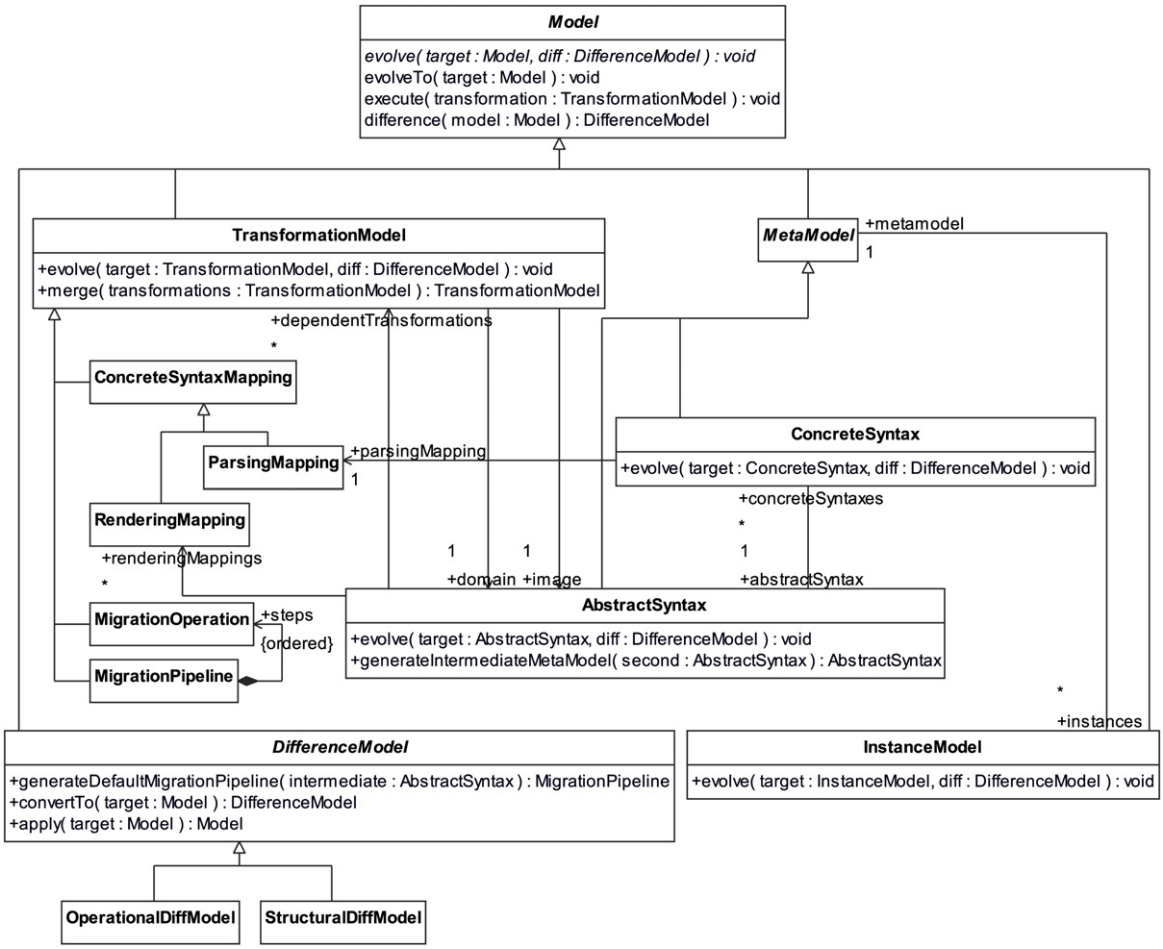


**Fig. 16.** Class Diagram of the different kinds of model artefacts in the framework.

Fig. 17 shows the method for concrete syntax evolution in the more complex free-hand editing. The difference is converted to the corresponding concrete syntax meta-model, and the algorithm of Fig. 16 is called. As a result, the meta-model evolves and will migrate related artefacts as described above.

Fig. 18 shows the method for transformation evolution in the more complex free-hand editing. Both image and domain can change through the changes of a transformation model. The difference model is converted to the domain meta-model, and the meta-model is evolved if there was a difference (lines 2–6). Analogously, the image is evolved (lines 7–11).

Fig. 19 shows the method for instance model evolution in the more complex free-hand editing. Analogously to Fig. 18, the difference model is converted to the meta-model and the meta-model is evolved.

Fig. 20 shows the method for evolution without the knowledge of the difference model. The previous algorithms are in principle only useable for tools that record differences and obtain the difference model automatically. Therefore, the method of Fig. 21 is included, where the difference is calculated by a difference algorithm, and the model is evolved, calling any of the other algorithms.

## 8. Conclusion and future work

The lack of support for automated evolution discourages necessary evolution of languages. In domain-specific modelling especially, modelling languages are used while under development or under on-going change. When such languages evolve, support for (semi-)automated evolution of language artefacts is needed. Previous research has been done only to support model co-evolution for meta-model evolution. Transformations or semantics are not yet taken into account. While addressing the problem of modelling language evolution, we made the following contributions in this paper:

- *taxonomy for migration*: we analysed related work, and a classification has been presented for migration of instance models;
- *syntactic/semantic evolution and consistency/continuity*: we divided up evolution into syntactic and semantic evolution. We identified a goal to satisfy consistency and continuity, effectively meaning that the evolution is syntactically correct

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

```
1: function void AbstractSyntax::evolve(AbstractSyntax target, DifferenceModel diff) do // overridden from Model
2:    MetaModel im = this.generateIntermediateMetaModel(target) // create intermediate meta-model
3:    MigrationPipeline pipe = diff.generateDefaultMigrationPipeline(im) // create migration pipeline
4:    pipe = INPUT(pipe) // manually adapt, combine and optimise operations in pipeline
5:    for all (MigrationOperation mm in pipe.steps()) do // migrate instance models
6:       for all (InstanceModel i in this.instances()) do
7:          i.execute(mm) // automatic or manual execution
8:       end for
9:    end for
10:   for all (TransformationModel t in this.dependentTransformations()) do // migrate transformations
11:      MigrationPipeline custompipe = INPUT(pipe) // manually adapt, combine and optimise operations in pipeline for each t
12:      t = INPUT(t) // the manual pass: semantic changes applied to the transformation model itself
13:      if (this == t.image()) then // image evolution
14:         t = t.merge(custompipe) // the transformation, followed by migration from intermediate model to new version
15:      else if (this == t.domain()) then // domain evolution
16:         t = custompipe.merge(t) // migration from new version to intermediate model, followed by the transformation
17:         for all (InstanceModel i in this.instances()) do // execute the transformation again
18:            i.execute(t)
19:         end for
20:      end if
21:   end for
22:   for all (ConcreteSyntax c in this.concreteSyntaxes()) do // migrate concrete syntaxes
23:      c = diff.convertTo(c).apply(c) // use homomorphic relation to convert diff and apply
24:      for all (RenderingMapping kappa in this.renderingMappings()) do // find rendering mapping for this concrete syntax to migrate
25:         if (kappa.image() == c) then
26:            kappa = diff.convertTo(kappa).apply(kappa) // use homomorphic relation to convert diff and apply
27:            break() // stop looking when it is found
28:         end if
29:      end for
30:      ParsingMapping kappaInv = c.parsingMapping() // get parsing mapping to migrate too
31:      kappaInv = diff.convertTo(kappaInv).apply(kappaInv) // use homomorphic relation to convert diff and apply
32:      for all (InstanceModel i in this.instances()) do // synchronise concrete syntax models with abstract syntax model
33:         i.execute(kappa)
34:      end for
35:   end for
36: end function
```

**Fig. 17.** The *evolve* method for *AbstractSyntax*.

```
1: function void ConcreteSyntax::evolve(ConcreteSyntax target, DifferenceModel diff) do // overridden from Model, assumes free-hand modelling
2:    AbstractSyntax as = this.abstractSyntax() // get the corresponding abstract syntax that will evolve
3:    DifferenceModel diffAs = diff.convertTo(as) // use homomorphic relation to convert diff
4:    AbstractSyntax asEvolved = diffAs.apply(as) // apply diff to obtain new version of meta-model
5:    as.evolve(asEvolved, diffAs) // evolve the abstract syntax
6: end function
```

**Fig. 18.** The *evolve* method for *ConcreteSyntax*.

```
1: function void TransformationModel::evolve(TransformationModel target, DifferenceModel diff) do // overridden from Model, assumes free-hand
   modelling
2:    AbstractSyntax domain = this.domain() // get the corresponding domain meta-model that will evolve
3:    DifferenceModel diffDom = diff.convertTo(domain) // use homomorphic relation to convert diff
4:    if not (diffDom is Null) then // check whether there was a change for the domain meta-model
5:       domain.evolve(diffDom.apply(domain), diffDom) // evolve the domain meta-model
6:    end if
7:    AbstractSyntax image = this.image() // get the corresponding image meta-model that will evolve
8:    DifferenceModel diffImg = diff.convertTo(this.image()) // use homomorphic relation to convert diff
9:    if not (diffImg is Null) then // check whether there was a change for the image meta-model
10:      image.evolve(diffImg.apply(image), diffImg) // evolve the image meta-model
11:   end if
12: end function
```

**Fig. 19.** The *evolve* method for *TransformationModel*.

```
1: function void InstanceModel::evolve(InstanceModel target, DifferenceModel diff) do // overridden from Model, assumes free-hand modelling
2:    MetaModel mm = this.metamodel() // get the corresponding meta-model that will evolve
3:    DifferenceModel diffmm = diff.convertTo(mm) // use homomorphic relation to convert diff
4:    mm.evolve(diffmm.apply(mm), diffmm) // evolve the meta-model
5: end function
```

**Fig. 20.** The *evolve* method for *InstanceModel*.

(*i.e.*, the conformance relation is preserved throughout the system) and semantically correct (*i.e.*, the system has evolved according to the changes that were intended);

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

```
1: function void Model::evolveTo(Model target) do // main method
2:     DifferenceModel diff = this.difference(target) // create difference model using a difference algorithm
3:     this.evolve(target, diff) // now that the difference is calculated, evolve the model
4: end function
```

**Fig. 21.** The *evolveTo* method for *Model*.

- *de/re-construction*: we deconstructed all possible (co-)evolution scenarios into four basic cases, which can each be handled (semi-)automatically. When applying such a basic scenario, we showed that the co-evolution of related artefacts, both instance models and transformation models, can be mapped onto these basic scenarios;
- *projection problem*: we identified the projection problem that prevents the full automation of transformation model migration by use of the migration transformation for instance models;
- *migration pipeline*: we introduced a flexible and modular migration pipeline, that can be reused for instance model migration as well as transformation model migration;
- *exploration of all possibilities*: we distinguished three levels of automation and we explored all possible consequences of modelling language evolution in a feature diagram, and discussed the level of automation possible;
- *algorithm*: we provided an algorithm that forms the backbone of our framework. The algorithm uses techniques that are introduced or discussed throughout the paper and constitutes a coherent solution for the problem of modelling language evolution.

We assume that an average MDE tool supports meta-modelling, transformation modelling and transformation execution. Following the discussion in this paper, however, the proposed approach depends on a few techniques. As these techniques are not yet featured in the average MDE tool, *maximally* automating all forms of evolution depends on the implementation of the following:

- *higher order transformation*: the automatic generation of migration transformations out of delta models requires support for higher order transformations, which are transformations that take other transformations as input and/or output. In order to support higher order transformations, the transformation language must be modelled explicitly (*i.e.,* the meta-model is not available). Several other uses for higher order transformation, inside and out of the context of evolution, are discussed in [5,30,33]. It is widely accepted that higher order transformations are a valuable feature in any MDE tool;
- *model differencing*: in order to support automated evolution on an industrial level, it must be possible to generate delta models out of two versions of a model. Moreover, it is desirable that the activity of meta-modelling does not have to change in order to support automated evolution. Difference tools such as DSMDiff [27] and UMLDiff [55] are useful as they detect differences retrospectively (see also Section 3.1);
- *transformation inverting*: in order to automatically co-evolve a transformation model in domain evolution, the inverse of the migration transformation is needed. This can be implicitly featured by providing the possibility to implement bidirectional transformations using Triple Graph Grammars (TGGs) [46]. However, in that case, one is restricted to the use of bidirectional transformation with triple graph grammars. It remains an open question whether TGGs are expressive enough to obtain the inverse of the migration transformation (which may delete elements). The other option is to automate the inversion of transformation models.
- *representation of semantics*: as not only the syntax but also the semantics of a modelling language evolves, there must be a way to represent these semantic changes. A more precise means to reason about semantics preservation is needed. Therefore, it is interesting to look more into the properties of a model, as suggested throughout this paper.
- *merging of transformations*: in order to optimise the sequences of transformations, they are merged. In this sense, automatic merging of transformations can be of great convenience. This can be simply done by chaining them, but the result can also be optimised.

The development of each of these techniques is a research topic in its own right. If all of these techniques are supported by an MDE tool of choice, a framework for evolution can provide maximal automation using the algorithm of Section 7.4. The framework is still valuable even if the prerequisites do not hold.

Future work on this subject will include elaborating the migration pipeline. Different restrictions on how to combine and order these operations will be investigated. Moreover, the special case of co-evolution of concrete syntax must be examined, as this can be optimised. There is a current trend of "weaving" language definitions together to create new DSMLs (*e.g.,* each individual language may focus on a specific concern, and a large system integration may need some links between these languages) [52]. We see this mainly in the Aspect-Oriented Modelling (AOM) community where such language weaving is considered a special kind of transformation [53]. We will look into the possibilities for optimisation.

Also, the role of semantics is very apparent when thinking about automation of language evolution. The precise modelling and analysis of semantics must be explored.

## Acknowledgements

We would like to thank the reviewers of this paper for their valuable comments. We would also like to thank the organisers and the participants of the Transformation Tool Contest 2010, Model Migration Case Study for the fruitful

discussions on the subject of this paper. Finally, the participants of the 2008–2010 Bellairs Computer Automated Multi-Paradigm modelling workshops are acknowledged for the stimulating discussions which have ultimately led to this paper. Partial support of this work by a discovery grant of the National Science and Engineering Research Council (NSERC) of Canada is gratefully acknowledged.

## References

[1] M. Alanen, I. Porres, Difference and union of models, in: UML'03: The Unified Modeling Language, 2003, pp. 2–17.
[2] J. Banerjee, W. Kim, H.-J. Kim, H.F. Korth, Semantics and implementation of schema evolution in object-oriented databases, ACM Special Interest Group on Management Of Data 16 (3) (1987) 311–322.
[3] F.P. Brooks, No silver bullet: essence and accidents of software engineering, IEEE Computer 20 (4) (1987) 10–19.
[4] K. Chen, J. Sztipanovits, S. Neema, Toward a semantic anchoring infrastructure for domain-specific modeling languages, in: EMSOFT'05: 5th ACM International Conference on Embedded Software, ACM, 2005, pp. 35–43.
[5] A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio, Automating co-evolution in model-driven engineering, in: EDOC'08: 12th International IEEE Enterprise Distributed Object Computing Conference, 2008, pp. 222–231.
[6] A. Cicchetti, D. Di Ruscio, A. Pierantonio, A metamodel independent approach to difference representation, Journal of Object Technology 6 (9) (2007) 165–185.
[7] J. De Lara, H. Vangheluwe, AToM$^3$: a tool for multi-formalism modelling and meta-modelling, in: Lecture Notes in Computer Science, vol. 2306, 2002, pp. 174–188.
[8] I. Galvao Lourenco da Silva, A. Goknil, Survey of traceability approaches in model-driven engineering, in: Eleventh IEEE International EDOC Enterprise Computing Conference, IEEE Computer Society Press, Los Alamitos, 2007, pp. 313–324.
[9] L. Geiger, A. Zündorf, Tool modeling with Fujaba, Electronic Notes in Theoretical Computer Science 148 (1) (2006) 173–186.
[10] H. Giese, T. Levendovszky, H. Vangheluwe, Summary of the workshop on multi-paradigm modeling: concepts and tools, in: T. Kühne (Ed.), Models in Software Engineering Workshops and Symposia at MoDELS 2006, in: Lecture Notes in Computer Science, vol. 4364, Springer-Verlag, 2006, pp. 252–262.
[11] B. Gruschko, D. Kolovos, R. Paige, Towards synchronizing models with evolving metamodels, in: International Workshop on Model-Driven Software Evolution at IEEE European Conference on Software Maintenance and Reengineering, ECSMR, 2007. http://www.sciences.univ-nantes.fr/MoDSE2007/.
[12] E. Guerra, J. de Lara, Model view management with triple graph transformation systems, in: International Conference on Graph Transformation, 2006, pp. 351–366.
[13] D. Harel, B. Rumpe, Meaningful modeling: what's the semantics of "semantics"?, IEEE Computer 37 (10) (2004) 64–72.
[14] R. Heckel, J. M. Küster, G. Taentzer, Confluence of typed attributed graph transformation systems, in: ICGT'02: First International Conference on Graph Transformation, Springer-Verlag, London, UK, 2002, pp. 161–176.
[15] J. Heering, R. Lämmel, Coupled software transformations, in: SET 2004, First Int. Workshop on Software Evolution Transformations, 2004, pp. 31–35.
[16] M. Herrmannsdoerfer, S. Benz, E. Juergens, Cope—automating coupled evolution of metamodels and models, in: 23rd European Conference on Object-Oriented Programming, ECOOP, 2009, pp. 52–76.
[17] J. Hoessler, J. Soden, Michael, H. Eichler, Coevolution of models, metamodels and transformations, Models and Human Reasoning (2005) 129–154.
[18] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, CMU, November 1990.
[19] S. Kelly, J.-P. Tolvanen, Domain-Specific Modeling: Enabling Full Code Generation, John Wiley & Sons, 2008.
[20] P. Klint, R. Lämmel, C. Verhoef, Toward an engineering discipline for grammarware, ACM Transactions on Software Engineering Methodology 14 (3) (2005) 331–380.
[21] T. Kühne, Matters of (meta-)modeling, Software and System Modeling 5 (4) (2006) 369–385.
[22] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, M. Wimmer, Explicit transformation modeling, in: Models in Software Engineering, Workshops and Symposia at International Conference on Model Driven Engineering Languages and Tools 2009, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 6002, Springer, 2010, pp. 240–255.
[23] R. Lämmel, Grammar adaptation, in: Formal Methods Europe (FME) 2001, in: Lecture Notes in Computer Science, vol. 2021, Springer-Verlag, 2001, pp. 550–570.
[24] R. Lämmel, W. Lohmann, Format evolution, in: 7th International Conference on Reverse Engineering for Information Systems, RETIS 2001, in: OCG, vol. 155, 2001, pp. 113–134.
[25] T. Levendovszky, D. Balasubramanian, A. Narayanan, G. Karsai, A novel approach to semi-automated evolution of DSML model transformation, in: 2nd International Conference on Software Language Engineering, SLE 2009, in: Lecture Notes in Computer Science, vol. 5969, 2010, pp. 23–41.
[26] X. Li, A survey of schema evolution in object-oriented databases, in: TOOLS'99: 31st International Conference on Technology of Object-Oriented Language and Systems, IEEE Computer Society, 1999, p. 362.
[27] Y. Lin, J. Gray, F. Jouault, DSMDiff: a differentiation tool for domain-specific models, in: Model-Driven Systems Development, European Journal of Information Systems 16 (4) (2007) 349–361 (special issue).
[28] T. Mens, S. Demeyer (Eds.), Software Evolution, Springer, 2008.
[29] T. Mens, P. Van Gorp, A taxonomy of model transformation, in: GraMoT'05, in: Electronic Notes in Theoretical Computer Science, vol. 152, 2006, pp. 125–142.
[30] B. Meyers, P. Van Gorp, Towards a hybrid transformation language: implicit and explicit rule scheduling in story diagrams, in: U. Assmann, J. Johannes, A. Zündorf (Eds.), Sixth International Fujaba Days, 2008, pp. 15–18.
[31] M. Minas, Generating meta-model-based freehand editors, Electronic Communications of the European Association of Software Science and Technology 1, 2006. http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/83.
[32] P.J. Mosterman, H. Vangheluwe, Computer automated multi-paradigm modeling: an introduction, Simulation 80 (9) (2004) 433–450.
[33] O. Muliawan, Extending a model transformation language using higher order transformations, in: Working Conference on Reverse Engineering, 2008, pp. 315–318.
[34] Object Management Group, February 2009, Unified Modeling Language Superstructure.
[35] Object Management Group, Business Process Modeling Notation (BPMN) Version 2.0, Tech. rep., OMG, 2010.
[36] Object Management Group, Object constraint language version 2.2. Tech. Rep., OMG, 2010.
[37] Object Management Group, OMG Unified Modeling Language Infrastructure version 2.3, May 2010.
[38] D. Ohst, M. Welle, U. Kelter, Differences between versions of UML diagrams, ACM Special Interest Group on Software Engineering 28 (5) (2003) 227–236.
[39] J. Peterson, Petri Net Theory and the Modeling of Systems, Prentice Hall, 1981.
[40] M. Pizka, E. Jurgens, Automating language evolution, in: TASE'07: First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, IEEE Computer Society, 2007, pp. 305–315.
[41] E. Rahm, P.A. Bernstein, An online bibliography on schema evolution, SIGMOD Record 35 (4) (2006) 30–31.
[42] M. Richters, M. Gogolla, A metamodel for ocl, in: UML'99: 2nd international Conference on The Unified Modeling Language, Springer-Verlag, 1999, pp. 156–171.

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002

[43] L. Safa, The practice of deploying DSM Report from a Japanese appliance maker trenches, in: J. Gray, J.-P. Tolvanen, J. Sprinkle (Eds.), Sixth Object-Oriented Programming, Systems, Languages and Applications Workshop on Domain-Specific Modeling, University of Jyväskylä, 2006, pp. 185–196.
[44] D.C. Schmidt, Guest editor's introduction: Model–Driven Engineering, IEEE Computer 39 (2006) 25–31.
[45] M. Schmidt, T. Gloetzner, Constructing difference tools for models using the SiDiff framework, in: International Conference on Software Engineering Companion'08: Companion of the 30th International Conference on Software Engineering, ACM, 2008, pp. 947–948.
[46] A. Schürr, Specification of graph translators with triple graph grammars, in: WG'94: 20th International Workshop on Graph-Theoretic Concepts in Computer Science, Springer-Verlag, London, UK, 1995, pp. 151–163.
[47] J. Sprinkle, G. Karsai, A domain-specific visual language for domain model evolution, Journal of Visual Languages and Computing 15 (3–4) (2004) 291–307.
[48] H. Su, D. Kramer, L. Chen, K. Claypool, E.A. Rundensteiner, Xem: managing the evolution of xml documents, in: RIDE'01: 11th International Workshop on Research Issues in Data Engineering, IEEE Computer Society, 2001, p. 103.
[49] A. A. Terekhov, C. Verhoef, The realities of language conversions, IEEE Software 17 (2000) 111–124.
[50] S. Vermolen, E. Visser, Heterogeneous coupled evolution of software languages, in: MoDELS'08: 11th international conference on Model Driven Engineering Languages and Systems, Springer-Verlag, 2008, pp. 630–644.
[51] G. Wachsmuth, Metamodel adaptation and model co-adaptation, in: 21st European Conference on Object-Oriented Programming, ECOOP'07, in: Lecture Notes in Computer Science, vol. 4609, Springer-Verlag, 2007, pp. 600–624.
[52] J. White, J.H. Hill, J. Gray, S. Tambe, A.S. Gokhale, D.C. Schmidt, Improving domain-specific language reuse with software product line techniques, IEEE Software 26 (2009) 47–53.
[53] J. Whittle, P. Jayaraman, MATA: a tool for aspect-oriented modeling based on graph transformation, in: H. Giese (Ed.), Models in Software Engineering, in: Lecture Notes in Computer Science, vol. 5002, Springer, Berlin/Heidelberg, 2008, pp. 16–27.
[54] M. Wimmer, A. Kusel, J. Schönböck, W. Retschitzegger, W. Schwinger, G. Kappel, On using inplace transformations for model co-evolution, in: Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL 2010), INRIA & Ecole des Mines de Nantes, 2010, 14 pages. URL: http://publik.tuwien.ac.at/files/PubDat_187033.pdf.
[55] Z. Xing, E. Stroulia, UMLDiff: an algorithm for object-oriented design differencing, in: ASE'05: 20th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2005, pp. 54–65.
[56] J. Zhang, J. Gray, A generative approach to model interpreter evolution, in: Object-Oriented Programming, Systems, Languages and Applications Workshop on Domain-Specific Modeling, 2004, pp. 121–129.

Please cite this article in press as: B. Meyers, H. Vangheluwe, A framework for evolution of modelling languages, Science of Computer Programming (2011), doi:10.1016/j.scico.2011.01.002