**Software and Systems Modeling manuscript No.**
(will be inserted by the editor)

# DSMCompare: Domain-Specific Model Differencing for Graphical Domain-Specific Languages

**Manouchehr Zadahmad[1], Eugene Syriani[1], Omar Alam[2], Esther Guerra[3], Juan de Lara[3]**

[1] Université de Montréal (Canada)
[2] Trent University (Canada)
[3] Universidad Autónoma de Madrid (Spain)

Received: date / Revised version: date

**Abstract** During the development of a software project, different developers collaborate on creating and changing models. These models evolve and need to be versioned. Over the past several years, progress has been made in offering dedicated support for model versioning that improves on what is being supported by text-based version control systems. However, there is still need to understand model differences in terms of the semantics of the modeling language, and to visualize the changes using its concrete syntax. To address these issues, we propose a comprehensive approach—called *DSMCompare*—that considers both the abstract and the concrete syntax of a domain-specific language (DSL) when expressing model differences, and which supports defining domain-specific semantics for specific difference patterns. The approach is based on the automatic extension of the DSL to enable the representation of changes and on the automatic adaptation of its graphical concrete syntax to visualize the differences. In addition, we allow for the definition of semantic differencing rules to capture recurrent domain-specific difference patterns. Since these rules can be conflicting with each other, we introduce algorithms for conflict resolution and rule scheduling. To demonstrate the applicability and effectiveness of our approach, we report on evaluations based on synthetic models and on version histories of models developed by third parties.

**Key words** Model-driven Engineering, Model differencing, Domain-specific languages, Graphical concrete syntax.

## 1 Introduction

Model-driven Engineering (MDE) relies on models to conduct all phases of software development. Models can be built using general-purpose modeling languages, e.g. UML, but the use of domain-specific languages (DSLs) is also common [28, 51].

Like other software artifacts involved in a development process, models evolve [48] and, therefore, need to be versioned to have a record of their changes [8]. Sometimes, models are persisted as text files (e.g., using the XML metadata interchange format, XMI [47]), which allows using code version control systems on them. However, text-differencing is not adequate for models as it may report irrelevant model differences (e.g., same objects that appear in different file positions). For this reason, the modeling community has proposed specific model versioning systems [3, 13, 25, 29] and approaches for model differencing [17], conflict resolution, and merging [10, 52].

An important aspect of a versioning system is the ability to visualize matches and differences of the history of a model in a comprehensible manner. However, many approaches, like EMFCompare [17], represent the differences between two versions of a model using low-level generic traces that may be difficult to understand. Moreover, these traces typically are at the abstract syntax level, which may further hinder their understanding, since users deal with models using their concrete syntax.

Therefore, we propose to represent traces in a domain-specific way, assign domain-specific semantics to recurring model differences (by defining semantic differencing rules), and visualize those differences at the concrete syntax level. Our approach lifts low level differences between two models to high level differences based on the semantics of the DSL and represents them by reusing the concrete syntax of the DSL. In this paper, we focus on graphical concrete syntaxes realized through the Sirius framework [54]. Since different semantic rules may conflict with each other, we propose an algorithm to assign priorities to rules, by their automated static analysis. To ensure the practicality of our proposal, we provide automated tool support to minimize the effort of applying the approach to arbitrary graphical DSLs.

The contributions of this paper are the following. First, we propose a method to represent model differences within a single domain-specific model. This is achieved by automatically extending the DSL meta-model with

domain-specific change operations. Second, we propose means to create higher-level representations of lower-level differences using semantic rules, provide mechanisms for analysing their possible conflicts, and propose scheduling policies for minimising those. Third, we provide an automated way to represent model differences using the graphical concrete syntax of the DSL. Finally, we provide a prototype tool support, able to adapt automatically Sirius-based editors for model change visualization, and use it to validate our approach on graphical DSLs and model histories created by third-parties.

This article extends our preliminary work [68] in several ways. First, we have made several improvements to the semantic differencing rules that encapsulate domain-specific differences. In Section 4.1, we explain how these rules can now express multiple negative application conditions. Also, the new Section 4.3 explains how the semantic differencing rules can be mapped to graph transformation rules. We illustrate our implementation using Henshin. This generalizes our approach, which now can be ported to other modeling frameworks. Second, in [68], we assumed that the rules are independent from each other and each rule is applied in isolation. However, in most scenarios, multiple semantic differencing rules may be applied on the same elements of a difference model. Therefore, we have devised an algorithm dedicated to resolving conflicting rules when they are applied in combination, which is presented in Section 5. The algorithm is directed to optimize the verbosity of the domain-specific difference model by suggesting a prioritized list of the rules to the user. Third, we extended the evaluation of our approach in several ways in Section 6. We have refined and extended the research questions to assess the effectiveness of our approach. To answer these, we now include a synthetic experiment, and validate our approach on two modeling projects developed by third-parties: Arduino[1] designer models and evolution of Ecore metamodels.

The rest of this paper is organized as follows. In Section 2, we overview the approach and introduce a running example. In Section 3, we describe how to represent model differences of a DSL. This is achieved by a semi-automated extension of the DSL metamodel and its concrete syntax. For the latter, we use Sirius as an illustration. In Section 4, we detail how to define high-level, domain-specific change descriptions in terms of semantic differencing rules. In Section 5, we explain how to resolve the conflicts when different rules are applied in combination. In Section 6, we evaluate the approach with one controlled experiment and two case studies. Finally, we discuss related works in Section 7 and conclude the paper in Section 8.

[1] `https://www.arduino.cc/`

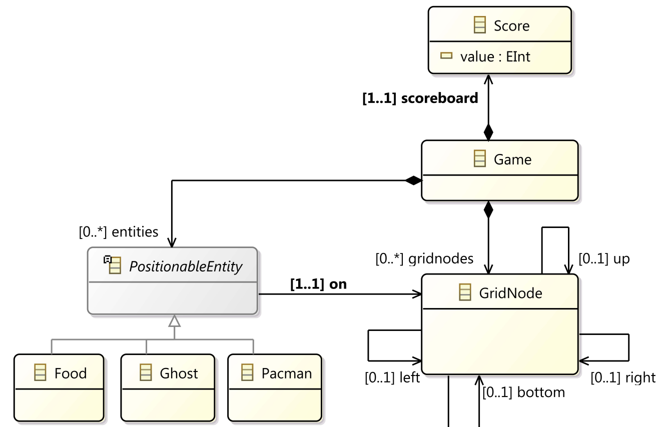

Figure 1: Metamodel of the Pacman game DSL

## 2 Overview and running example

In the following, we motivate our approach with a running example and present its overall rationale.

### *2.1 Motivating example*

A typical model differencing tool compares two versions of a model based on the performed editing steps (e.g., added class or deleted reference). The result of this comparison is identified by low-level differences between the two versions, which includes at least two sets: *match* and *diff*. The match set establishes a pair-wise correspondence between similar elements in both models. The diff set computes the differences between each pair in the match set. The most popular generic model comparison tools, EMFCompare [17] for instance, produce three kinds of diffs: ADD, DELETE, and MODIFY.

However, a DSL user works with an end-user tool and does not interact with the abstract syntax. Instead, she uses end-user features such as domain-specific views and diagrams to manipulate models. Any change in this level of abstraction (i.e., the domain-specific concrete syntax) can turn into several fine-grained changes in the model. Consequently, the comparison tool shows the user all low-level changes, such as a deleted reference between two objects, which may not make sense to a DSL user who is not familiar with the metamodel of the DSL. This creates a mismatch between what the comparison tool produces and what a DSL user would expect to understand: the differences in terms of domain-specific syntax rather than concepts of the abstract syntax.

There have been approaches that tried to mitigate this issue, e.g., through the semantic lifting of the low-level changes [26] or by using a metamodel to represent model differences [14]. However, these approaches do not provide a comprehensive framework for handling domain-specific model differences. In particular, the existing approaches mostly focus on expressing model differences at the abstract syntax level and do not show differences at the concrete syntax level (i.e., the graphical notation

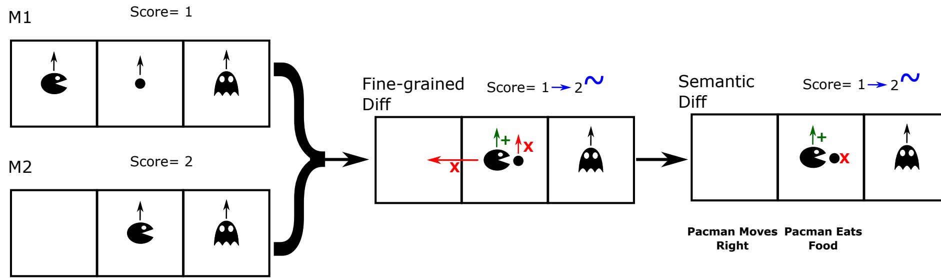


Figure 2: Running example using DSMCompare

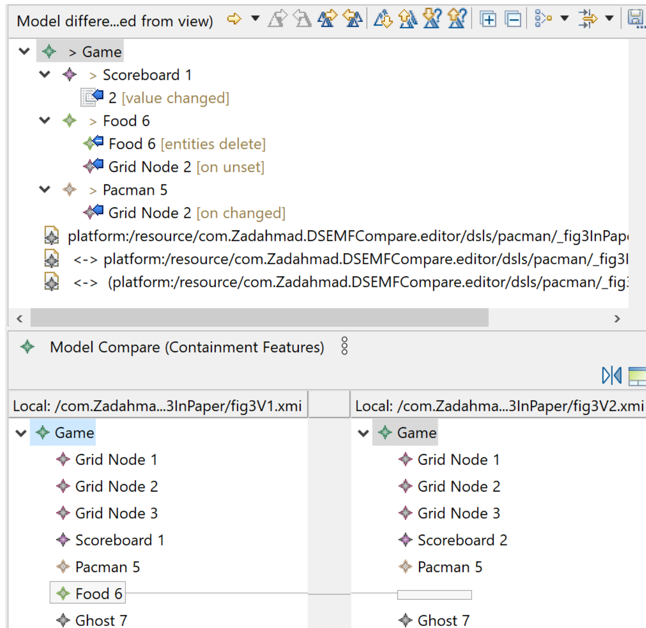


Figure 3: Representation of difference model in EMF-Compare for the Pacman game DSL

of a DSL). Furthermore, the existing approaches do not take domain-specific model semantics[2] into consideration during the comparison process.

To address these issues, we introduce an approach, called *DSMCompare*, which provides the DSL user with a set of *semantic domain-specific model differences* that highlight the differences between two versions of a model at both the abstract and concrete syntax levels. We explain how a DSL user uses DSMCompare using a running example of a simplified Pacman game, a well-known game where Pacman navigates through grid nodes searching for food to eat, while ghosts try to kill him.

We provide a modeling environment to define game configurations, based on [59]. Figure 1 shows the metamodel of this game. Figure 2 sketches what DSMCompare outputs given two versions (M1 followed by M2) of a Pacman game configuration. The black arrows pointing up over Pacman, food, and the ghost are the associations representing their position on a grid node. Comparing M1 and M2, we can easily conclude that Pacman has moved right to the middle grid node and ate the food on it. The score value is incremented accordingly. DSMCompare produces a domain-specific difference model $Diff_{12}$ in two steps. First, in the middle of Figure 2, $Diff_{12}$ contains all the fine-grained diffs. The green arrow with a '+' denotes that an association is added to a grid node, the red arrow with an 'x' denotes a deleted association, and the blue arrow with a ∼ (on the scoreboard) denotes an attribute value change. Then, DSMCompare applies the provided semantic differencing rules on $Diff_{12}$. In this case, two rules can be applied: *Pacman Eats Food* and *Pacman Moves Right*. For example, the former rule checks that Pacman is on a grid node that also has food on it which gets deleted, and the scoreboard value is incremented. The final difference model $Diff_{12}$ is depicted at the right of Figure 2 (labelled as *semantic diff*).

In contrast, using EMFCompare for comparison results in a list of low-level changes as presented in Figure 3. The DSL user needs additional analytical effort to understand these changes to infer the difference in a meaningful way. For example, the user needs to understand that (on the top panel of Figure 3) "on changed" means that Pacman has moved to a different grid node (because the reference "on" has changed), and needs to inspect the lower juxtaposed panels to understand that food has disappeared. However, as the "on" reference is not shown on the tree editor, it becomes difficult to realize that this is because Pacman ate the food.

### *2.2 Overview of DSMCompare*

Figure 4 gives an overview of DSMCompare. The approach is useful for two types of users: DSL engineers (who build the DSL) and DSL users (who create models using the DSL).

To define the DSL, the engineer creates a metamodel *MM* for the abstract syntax, and a model *CS* of the concrete syntax. In DSMCompare, we reuse both components to define the domain-specific model differences for that DSL and show any domain-specific diff $Diff_{12}$ between two versions of a model *M1* and *M2*. Concretely, the

[2] In this paper, we use "domain-specific model semantics" to refer to the meaning that a human assigns to a model when looking at it, not to the execution semantics of the model.

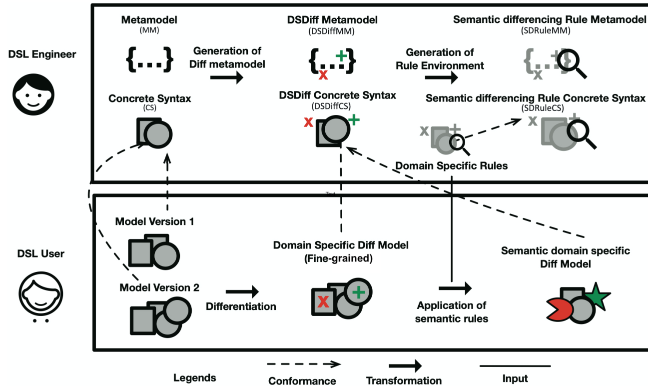


Figure 4: Overview of the approach

approach produces a domain-specific diff metamodel *DSDiffMM* and concrete syntax model *DSDiffCS*, as shown in Figure 4. *DSDiffMM* extends the language metamodel to define domain-specific diffs, such as adding/removing a model element. *DSDiffCS* shows the corresponding concrete syntax elements: graphical elements that could be added, removed, or updated.

DSMCompare also produces an environment to describe high-level semantic differences in the form of rules tailored to the DSL. Namely, it produces a semantic differencing rule metamodel *SDRuleMM* and concrete syntax model *SDRuleCS*, to allow the DSL engineer to define the set of rules to apply on $Diff_{12}$. As discussed previously, having semantic differencing rules is important to facilitate reasoning about model differences. Without these rules, low-level differences may not convey the intention or the reason behind a change, and it may be difficult to understand for the user how changes relate to each other. For example, the DSL engineer could define a rule for *operation overriding* in class diagrams, which matches an operation in one version of a model with a variant of that operation in a different version. Instead of showing that an operation is simply being added in the second version, DSMCompare uses the rule to represent this change as the second operation overriding the first one.

The DSL user can use DSMCompare for different purposes. For example, in a version control system, the DSL user may want to understand high-level semantic differences between two versions of a class diagram. By using rules that represent refactorings, it would be possible to identify the places in the model that underwent refactoring. In a collaborative development environment, a DSL user may identify the domain-specific changes that a collaborator introduced, by applying DSMCompare on the collaborator version of the model and the model at hand.

DSMCompare produces a traditional diff of the two model versions by reusing a difference tool such as EMF-

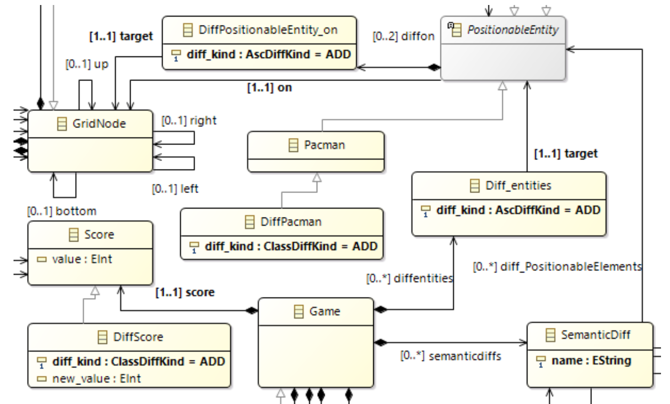


Figure 5: Excerpt of the generated difference metamodel

Compare. This result is processed to generate $Diff_{12}$, that conforms to *DSDiffMM*, and is represented according to *DSDiffCS*. At this point, $Diff_{12}$ contains the fine-grained differences in the concrete syntax of the DSL. With a library of rules predefined by the DSL engineer, the approach executes the applicable rules on $Diff_{12}$ to produce a semantically lifted difference model.

## 3 Fine-grained differencing

To overcome the restrictions of generic approaches for model comparison, we propose to represent all model differences in a format tailored to the domain of the original metamodel. We also visualize the differences using domain-specific concrete syntax.

Section 3.1 explains how to extend the domain metamodel (*MM*) to represent two model versions within one model. Then, Section 3.2 describes how the concrete syntax model (*CS*) is extended to represent model changes (*DSDiffCS*). Finally, Section 3.3 introduces how a single diff model $Diff_{12}$ (instance of *DSDiffMM*) is generated out of two model versions (*M1* and *M2*), and how this is represented using the diff concrete syntax *DSDiffCS*.

### *3.1 Domain-specific difference metamodel*

To represent model differences in a domain-specific way, the metamodel of model differences should remain faithful to the original metamodel *MM*. Therefore, we create a new metamodel *DSDiffMM* for domain-specific differencing (see Figure 5) based on *MM* (see Figure 1).

Algorithm 1 outlines the transformation from *MM* to *DSDiffMM*. It starts by cloning *MM* to ensure that *DSDiffMM* comprises all the structural features of the DSL. In Figure 5, *DSDiffMM* includes all classes and associations that the *MM* metamodel possesses. The remaining steps extend the metamodel as follows. We create two enumerations that will be used to annotate each class and association with the kind of difference. To represent a difference in an object of a class, like `Score`, we create a subclass with an additional attribute

**Algorithm 1** Transformation from MM to DSDiffMM

```
1: procedure GenerateDSDiffMM(MM)
2:   DSDiffMM ← MM.clone( "DSDiffMM")
3:   DSDiffMM.createEnum( "ClsDiffKind", {ADD,DEL,MOD})
4:   DSDiffMM.createEnum( "AscDiffKind", {ADD,DEL})
5:   for all class C in DSDiffMM do
6:     if not C.isAbstract() then
7:       DiffC ← DSDiffMM.createClass( "Diff"+C)
8:       DiffC.setSuperClass(C)
9:       DiffC.addAttribute( "diff_kind", ClsDiffKind)
10:    end if
11:    for all attribute a in C.getAllUniqueAttributes() do
12:      DiffC.addAttribute( "new_"+a, a.getType())
13:    end for
14:  end for
15:  for all association S in DSDiffMM do
16:    C1 ← S.getSource(), C2 ← S.getTarget()
17:    if C1 ≠ DSDiffMM.getRootClass() then
18:      DiffC1_S ← DSDiffMM.createClass( "Diff"+C1+ "_"+S)
19:      DiffC1_S.addAttribute( "diff_kind", AscDiffKind)
20:      n ← S.getTargetCardinalities().target().upperBound()
21:      if S.isComposition() then
22:        diffS ← C1.addComposition( "diff"+S, DiffC1_S)
23:      else
24:        diffS ← C1.addAssociation( "diff"+S, DiffC1_S)
25:      end if
26:      diffS.setCardinalities(1..1, 0..2×n)
27:      target ← DiffC1_S.addAssociation( "target", C2)
28:      target.setCardinalities(0..1, 1..1)
29:    end if
30:  end for
31:  SDiff ← DSDiffMM.createClass( "SemanticDiff")
32:  SDiff.addAttribute( "name", String)
33:  for all class C in DSDiffMM do
34:    diff_C ← SDiff.addAssociation( "diff_"+C, C)
35:    diff_C.setCardinalities(1..1, 0..*)
36:  end for
37:  R ← DSDiffMM.getRootClass()
38:  diffs ← R.addComposition( "diff"+S, SDiff)
39:  diffs.setCardinalities(1..1, 0..*)
40:  return DSDiffMM
41: end procedure
```

`diff_kind` that states whether the object has been added, deleted, or that at least one of its attributes has been modified. In the subclass we also add, for each attribute in the class, a new attribute of the same type that will hold the new value. For example, the subclass of `Score` has an attribute `new_value`. This is particularly useful when auditing changes in different versions of a same model.

Note that this procedure does not transform the class inheritance hierarchies. If *MM* has a class `A` and a class `B` that inherits from `A`, then, in *DSDiffMM*, `DiffA` inherits from `A` and `DiffB` inherits from `B`, but there is no inheritance between `DiffA` and `DiffB`. We argue that this decision is to allow implementing our solution in frameworks where multiple inheritance is not supported. Therefore, on line 11 of Algorithm 1, `C.getAllUniqueAttributes()` retrieves all attributes of `C` and those inherited from its super classes transitively. Furthermore, abstract classes have no corresponding *Diff* class since they cannot be instantiated in the compared models.

As outlined in lines 15–30 of Algorithm 1, for each association in *MM*, we create a class to reflect the kind of change (addition or deletion). We then connect this new class with the source and target classes of the association. In the Pacman example, the `on` association is transformed into the `DiffPositionableEntity_on` class. Since `on` is a composition, `diffon` is also a composition, to preserve the semantics of the association. Suppose that a difference model $Diff_{12}$ needs to reflect that the Pacman object has moved from one grid node to another. Then, there will be two `DiffPositionableEntity_on` instances: one representing the deletion of the `on` relation to the old grid node and one for the addition of the `on` relation to the new grid node. This is why the upper bound of the cardinality of `diffon` in *DSDiffMM* must be doubled on line 26.

The elements created up to now can only capture individual fine-grained differences in $Diff_{12}$. To enable the representation of semantic differences, the procedure creates a `SemanticDiff` class (cf. line 31) that holds the name of the semantic difference that a combination of original and semantic diff classes represent. This will be used by DSMCompare in the second step when applying semantic differencing rules (cf. Section 4).

One benefit of this procedure is that a difference class, like `DiffScore`, still contains all attributes and relations with the same name, type, cardinalities, and constraints as in `Score`. The rationale is to allow an instance of *MM* to be a valid instance of *DiffMM*. This is useful in case *M1* and *M2* are identical, as their difference can be represented by *M1*. Consequently, a difference model can contain both instances of `Score` and `DiffScore` if one is unchanged and the other is, say, deleted.

### *3.2 Visualization of domain-specific differences*

Since the user of the DSL manipulates models in their concrete syntax representation, it makes no sense for her/him to analyze the difference model in its abstract syntax form. Therefore, the DSL to represent the difference model should also be assigned a concrete syntax, which we call *DSDiffCS*. Since the DSL engineer has defined a concrete syntax *CS* for the DSL, she should also provide one for *DSDiffMM*. However, instead of starting from scratch, we propose to generate a default *DSDiffCS* that reuses the style from *CS* to remain in the spirit of the DSL. Then, the DSL engineer can customize it if so desired. In this subsection, we describe how to generate *DSDiffCS* from *CS*, assuming a graphical concrete syntax.

Sirius [54] is one of the most popular frameworks to generate graphical modeling environments and to manipulate models graphically in the Eclipse ecosystem.

Although our approach is applicable to other graphical language workbenches, such as GMF [22], MetaEdit+ [27] and AToMPM [60], our description is based on Sirius because its wide use nowadays, and because it offers a model-based approach for concrete syntax definition.

In Sirius, the main component of the concrete syntax definition is a viewpoint specification model (`odesign`). It defines a mapping of graphical representations to elements of *MM*. For example, to render the visualization of the `Pacman` class, we define a `NodeMapping` that refers to an icon in an image file. The `NodeMapping` can be a combination of text, icons, shapes and style customizations, such as color and size. Similarly, associations are rendered by an `EdgeMapping`. As for compositions, the target class is rendered by a `BorderedNodeMapping` within the `NodeMapping` of the source class. Constraints expressed in the Acceleo Query Language (AQL), a variant of the Object Constraint Language (OCL) [46], can filter visualizations depending on a condition. Finally, it is possible to define a palette of buttons to instantiate *MM* classes and associations by customizing the `ToolSection`.

We generate *DSDiffCS* by means of an outplace transformation[3] that takes as input *CS* and outputs *DSDiffCS*. The overall logic of the transformation is to copy each component of *CS* onto *DSDiffCS* and create the representation of each `Diff_` class by extending the representation of its corresponding *MM* class. This maximizes the reuse of *CS* to represent the difference model intuitively for the DSL user. For each `NodeMapping`, e.g., `PacmanNode`, we create three new ones for each difference kind: `DiffPacmanNodeADD`, `DiffPacmanNodeDELETE`, `DiffPacmanNodeMODIFY`. By default, the add node is the same as the original node annotated with a green '+' sign, the delete with a red 'x', and modify with a blue '∼'. The latter indicates that at least one of the attribute values has changed. For example, the `ScoreNode` is a rectangle with the value of its `value` attribute displayed inside. We change the text displayed in `DiffScoreNodeMODIFY` by showing the `value` concatenated with an arrow '−>', followed by the `new_value`. One particularity of the mapping in Sirius is that if `DiffPacman` inherits from `Pacman` in *DSDiffMM*, Sirius displays the representation of the former for the latter. Therefore we need to add an AQL condition in `DiffPacmanNodeADD` to force it to represent `DiffPacman` instances only and not its super classes.

`EdgeNodes` are treated slightly differently. Recall that an association `S` from class `A` to class `B` in *MM* is transformed into a class `DiffA_S` with an incoming composition `diffS` from `A` and an outgoing association `target` to `B`. Therefore, in *DSDiffCS*, `DiffA_S` is represented with a `BorderedNodeMapping` as a subnode of the `NodeMapping` of `A`. We create two `BorderedNodeMappings` for each `Edge`, one for adding and one for deleting, annotated similarly to `Nodes`. The `target` association is rendered by an `EdgeNode`.

[3] This is a transformation that takes as input a model and produces a different output model. This contrasts with inplace transformations, which are applied directly on the input model.

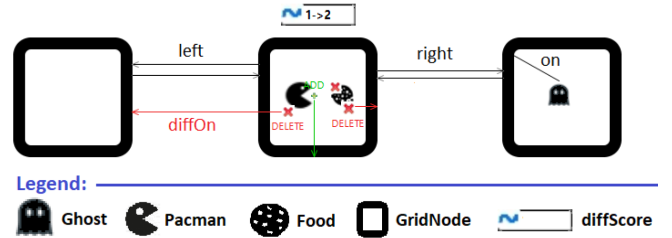

Figure 6: Fine-grained difference model $Diff_{12}$ of *M1* and *M2*

The only element in *DSDiffMM* that does not have a visualization in *CS* is the `SemanticDiff` class (cf. line 31 of Algorithm 1). By default, we represent it with a rectangle with its `name` attribute value displayed inside.

We implemented this transformation in ATL to help automate the process. If the concrete syntax makes use of icon files to render the elements of the metamodel, the DSL engineer must also provide a set of icon files for each `Diff_` class and association. The transformation assumes that the name of the icon is preserved, but suffixed with the `DiffKind`, e.g., `pacman.png` → `pacman_add.png`. Nevertheless, it is also possible to fully automate that part if the concrete syntax does not include external icons, but is built entirely with Sirius nodes. In this case, our transformation will automatically add a symbol on the top-left of the node indicating the `DiffKind`. This opens the door to a variety of visualizations to represent domain-specific semantic differences.

Defining *DSDiffMM* along with *DSDiffCS* as a domain-specific difference language using frameworks such as Sirius, allows the DSL engineer to generate a domain-specific model environment to represent difference models $Diff_{12}$. These can be inspected and manipulated like any other model (*M1* and *M2*) in an environment familiar to the DSL user. Figure 6 illustrates the $Diff_{12}$ model for the running example (cf. Figure 2), presented in its concrete syntax as output by DSMCompare.

### *3.3 Fine-grained domain-specific model comparison*

Given two models *M1* and *M2* of a DSL, we want to output a single model $Diff_{12}$ depicting the changes from *M1* to *M2*, as an instance of *DSDiffMM*. Note that the two models are provided with their abstract and concrete syntax representations. Most current model comparison approaches detect changes at the abstract syntax level only. For instance, [36] dynamically computes an identifier for each model element based on their properties (e.g., type and attribute values). Alternatively, metamodel-agnostic approaches, like [12, 14], compute the structural and attribute value similarities between *M1* and *M2*. These tools produce a generic difference model that lists the changes between the two models. We chose to reuse these

difference algorithms and then process the result to produce $Diff_{12}$. In our implementation, we rely on the change list output by EMFCompare.

To produce the $Diff_{12}$ model, we first clone *M1* since the differences will be expressed in terms of *M1*. We assume that the result from a difference algorithm outputs a list $\Delta_C$ of differences for classes, and another one $\Delta_A$ for associations, such as the case in EMFCompare. We denote an element $E' \in \Delta_C$ using primed uppercase letters. This way, if $E'$ is a deletion or a modification, we identify $E$ to be the corresponding element in *M1*. For example, in Figure 6, $E'$ can be the score object with its value modified from 1 to 2. We replace $E'$, the score object, in *M1* by an instance of the `DiffScore` class as per Algorithm 1. This new object will hold all original attribute values, so `score=1`, and all new attribute values, so `new_score=2`. If $E'$ is an addition, we create an instance of the Diff class corresponding to $E'$ and set all its new attribute values. Finally, we mark the new Diff element with its `ClassDiffKind`.

An association $A' \in \Delta_A$ is treated a bit differently. If $A'$ is a deletion, we remove the link $A$ in *M1* corresponding to $A'$ and create an instance of the Diff class corresponding to it. For example, in Figure 6, the `on` link from the Pacman to the first grid node is removed and an instance of `DiffPositionableElement_on` is created. In case $A'$ is an addition, only the creation of the Diff class is needed. We then connect the Diff instance to the source and target elements of $A$. Finally, we mark it with its `AscDiffKind`.

Our approach does not require additional manual effort to produce the concrete syntax of $Diff_{12}$. Since $Diff_{12}$ is an instance of *DSDiffMM*, then *DSDiffCS* is applied automatically on $Diff_{12}$ to represent it visually, as shown in Figure 6.

## 4 Domain-specific semantic differencing

In this section we introduce the approach to create semantic diff rules. This involves synthesizing a metamodel *SDRuleMM* out of *DSDiffMM*, as we explain in Section 4.1. Then, in Section 4.2 we outline how to generate a graphical environment for the DSL engineer that supports the creation of semantic differencing rules, based on *SDRuleMM*. Finally, Section 4.3 provides a semantics for domain-specific diff rules in terms of graph transformation rules [16].

### *4.1 Rules for domain-specific differences*

As explained in Section 2, we automatically derive an environment for specifying semantic differencing rules. This enables the DSL engineer to define higher-level changes specifically tailored for the domain. A rule needs to detect a pattern of fine-grained differences and replace it with a `SemanticDiff` class that was created in Algorithm 1.

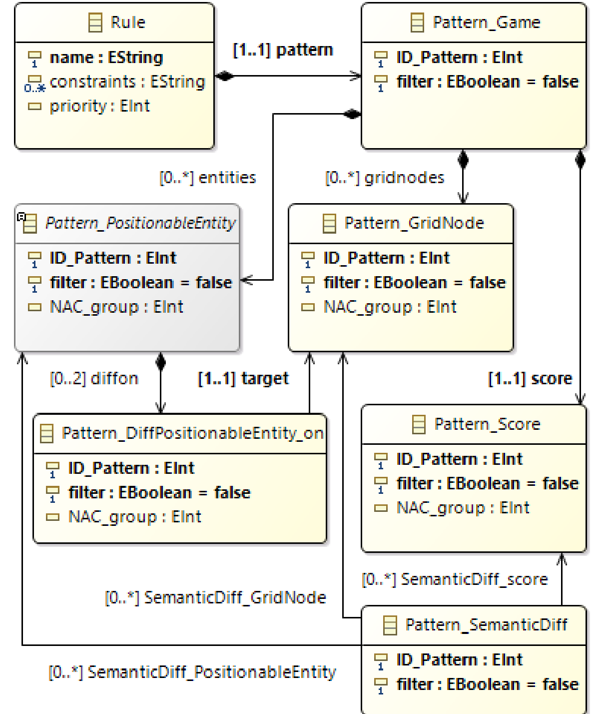


Figure 7: Excerpt of the semantic differencing rule metamodel *PacmanRuleMM*

Our semantic differencing rules act similarly to inplace model transformation rules [16] with a precondition and a postcondition component. Algorithm 2 outlines the procedure to produce *SDRuleMM* from *DSDiffMM* and Figure 7 shows the result. It is inspired by [31] where the authors produce domain-specific model transformation rule patterns from a DSL.

Like Algorithm 1, this procedure starts by reusing all the elements of *DSDiffMM*, adapting them to the new needs. Every class and association is prefixed with `Pattern_`, except the `SemanticDiff` class. All attributes from *DSDiffMM* except `diff_kind` are removed, since they do not contribute to the rule. However, the connectivity of the associations remains as in *DSDiffMM*. This simplifies the detection of patterns in the difference model $Diff_{12}$.

We add two attributes to all pattern classes. First, a unique identifier distinguishes instances of the same classes to facilitate writing constraints. Then, a *filter* attribute is used to signify that the element in $Diff_{12}$ should be removed when applying the rule. It is helpful to remove fine-grained differences when a domain-specific difference is more meaningful. Furthermore, the rule may contain negative application conditions (NACs) to forbid the presence of elements [16]. We add a `NAC_group` attribute to all classes prefixed with `Pattern_`. Similar to some transformation languages [5], one or more rule elements set with the same `NAC_group` value constitute a NAC. Multiple values of this attribute are used to represent several NACs in the rule, none of which can be matched for the rule to be applicable.

**Algorithm 2** Transformation from `DSDiffMM` to `SDRuleMM`

```
1: procedure GENERATESDRULEMM(DSDiffMM)
2:   SDRuleMM ← DSDiffMM.clone( “SDRuleMM”)
3:   for all class C≠SemanticDiff in SDRuleMM do
4:     C.keepDiffKindAttribute()
5:     Pattern_C ← C.setName( “Pattern_” + C.getName())
6:     Pattern_C.addAttribute( “ID_Pattern”, int)
7:     Pattern_C.addAttribute( “filter”, bool)
8:     Pattern_C.addAttribute( “NAC_group”, int)
9:   end for
10:  for all association S in SDRuleMM do
11:    S.setName( “Pattern_” + S.getName())
12:  end for
13:  Rule ← SDRuleMM.createClass( “Rule”)
14:  Rule.addAttribute(“name”, String)
15:  Rule.addAttribute(“constraints”, String[])
16:  Rule.addAttribute(“priority”, int)
17:  R ← SDRuleMM.getRootClass()
18:  pattern ← Rule.addComposition( “pattern”, R)
19:  pattern.setCardinalities(1..1, 1..1)
20:  return SDRuleMM
21: end procedure
```

Finally, lines 13–16 of the algorithm add a new `Rule` class as the new root of the metamodel. This enables the transformation engine to navigate easily through the elements of the rule. In addition, the `Rule` class allows specifying a list of constraints over attribute values. In practice, constraints are written in Java and executed dynamically using BeanShell[4], an embedded interpreter to run Java scripts. Within constraints, pattern objects (elements of the rule) can be accessed through the `Item` keyword, using their identifier and the desired attribute name in the form of `Item(ID,[ATTR_NAME])`. Figure 8 shows an example semantic rule called `Eat` (in concrete syntax) with a constraint. This constraint states that the new value of the score should be greater than the original value for the rule to be applicable.

### *4.2 Automatic generation of a graphical environment for semantic diff rules*

Our approach not only helps the DSL user to better understand the difference between two models, but it also assists the DSL engineer to design conveniently the semantic differencing rules in the same language workbench.

For this purpose, we automatically generate a concrete syntax for rules (called *SDRuleCS*) out of the *DSDiffCS* model by a transformation. The transformation is very similar to the one described in Section 3.2. First, we copy the viewpoint specification model and adapt it to *SDRuleMM*. Each `NodeMapping` displays ≪filter≫ if the `filter` attribute is set to true, as well as the `ID_Pattern` of the object. All other attribute values from their *DSDiffMM* counterparts are removed as they are no longer present in pattern classes, like in the `Score`. To create and edit a rule, the DSL designer is provided with a palette showing all rule-specific elements, including those from *DSDiffCS*.

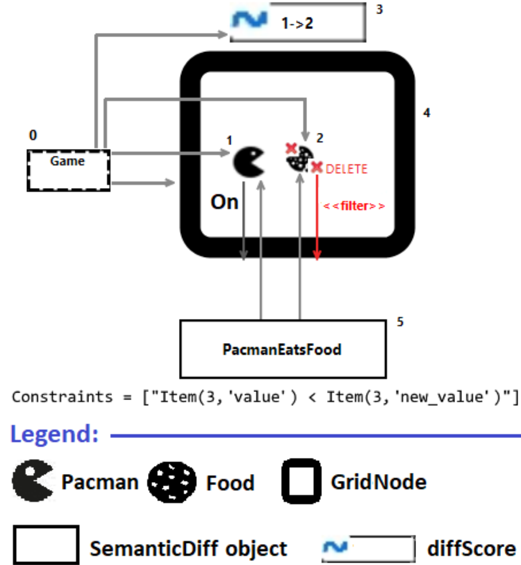


Figure 8: The semantic differencing rule `Eat`, abstracting fine-grained differences to depict that Pacman has eaten food

Figure 8 illustrates a rule in the generated domain-specific environment. The rule describes that a *Pacman eat food* change occurs when Pacman is on a grid node, a food is deleted from the same node, and the score is incremented. To reduce the amount of fine-grained differences reported to the DSL user, the rule also filters the `on` association from the food to the grid node.

### *4.3 Executing the semantic diff rules*

As outlined in Figure 4, we apply the semantic diff rules to enhance the fine-grained difference model $Diff_{12}$ with semantic differences, and possibly remove fine-grained differences. Given the difference model $Diff_{12}$ produced as described in Section 3.3, we apply the rules on $Diff_{12}$ as an inplace model transformation. For this purpose, we express the semantics of our semantic diff rules as graph transformation rules. In particular, we use Henshin [5] as the target transformation engine. Henshin is an inplace model transformation language implementing graph transformations for the Eclipse Modeling Framework. Therefore, we opted to transform each SDRule into a semantically equivalent Henshin rule, which can then be applied on $Diff_{12}$. In practice, we implemented this higher-order transformation using an Xtend-based code generator. This takes a set of semantic differencing rules and produces a set of Henshin rules. We chose a code generator approach since Henshin rules can be specified in a textual notation [58].

In a semantic differencing rule SDRule, the precondition consists of the constraints of the rule and the

[4] `https://github.com/beanshell/beanshell`

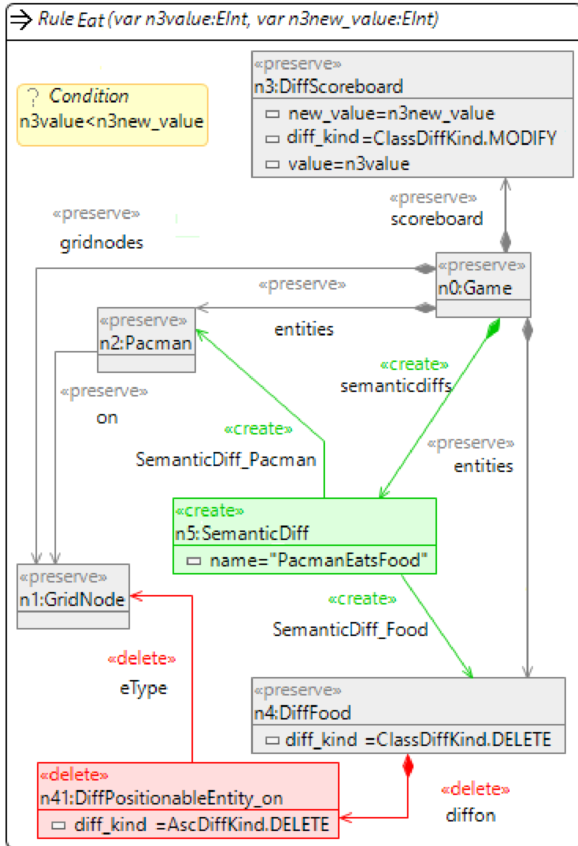

Figure 9: Rule `Eat` transformed into Henshin

structure formed by the pattern objects (typed by a class prefixed with `Pattern_`) contained inside the rule except for the `SemanticDiff` object. The postcondition of the rule is specified by the `SemanticDiff` instance and its `diff_` associations (see lines 31–36 of Algorithm 1), along with all `filter` attributes that are set to `true` in the pattern classes.

For example, the `Eat` rule in Figure 8 looks for a `Pacman` object and a deleted `DiffFood` on the same grid node. It also requires that the new value of `DiffScore` has increased. Then, it creates the `SemanticDiff` object named `PacmanEatsFood` and hides the deleted `DiffPositionableElement_on` link associated with `DiffFood`. Figure 9 shows how this rule is encoded in Henshin. A Henshin rule HRule consists of nodes, edges, and conditions. Nodes and edges can be assigned actions (preserve, create, delete, forbid) and are typed by a metamodel class or association respectively. Nodes can have attribute values.

Algorithm 3 presents the transformation from SDRule to HRule. We briefly outline the transformation steps to create an HRule from a SDRule in what follows:

1. Create an HRule with the same name as the SDRule (line 2 of Algorithm 3).
2. Create a condition in HRule for every condition in SDRule. If a condition uses an attribute, add a parameter to the rule, then assign the parameter to the corresponding attribute and use the parameter instead of the attribute in the condition (lines 4–8).
3. Create a node with action ≪preserve≫ in HRule for every pattern object with no filter and no `NAC_group` set in SDRule (lines 11–13).
4. Create a node with action ≪delete≫ in HRule for every pattern object with filter set to true in SDRule (lines 14–15).
5. Create a node with action ≪forbid≫ in HRule for every pattern object with a `NAC_group` set in SDRule. Set the forbid identifier to the value of the NAC group (lines 16–18).
6. Create a node with action ≪create≫ in HRule for every `SemanticDiff` object in SDRule (lines 19–20).
7. If a pattern object has a value for its attributes like `diff_kind` set in SDRule, create the same attribute with the same value in the corresponding Henshin node (lines 22–26).
8. Create an edge in HRule for each association in SDRule. The type of the edge should correspond to the one of the association (lines 28–41) as follows. All edges adjacent to a node of type `SemanticDiff` have the action ≪create≫ (lines 34–35). All edges adjacent to a node with action ≪delete≫ or ≪forbid≫ have also the action ≪delete≫ or ≪forbid≫ respectively (lines 36–39). Otherwise, the edge action is set to ≪preserve≫ (lines 40–41).

Thanks to the transformation to Henshin, our rules support matching a subclass of a pattern class [7]: in *DSDiffMM*, the `DiffScore` class inherits from the `Score` class. Furthermore, abstract classes from *MM*, like `PositionableElement`, can be used when specifying patterns, which can be useful to define fewer rules [15].

To apply all the semantic differencing rules with Henshin, we must set the control flow of the transformation. For this purpose, we group all HRules inside an *independent unit* so that all rules are applied in an arbitrary order nondeterministically. Furthermore, each HRule is executed in a *loop unit* so that each rule is applied iteratively as long as matches are found before any other rule is applied. When the transformation execution concludes, all objects marked as filtered in the pattern are removed and objects semantically meaningful to the domain are added to the difference model. Altogether, the resulting $Diff_{12}$ model is *semantically lifted* to show higher-level differences that are deemed important and meaningful to the DSL user. Moreover, lower-level (fine-grained) differences may be deleted by the rule, hence reducing verbosity. Applying the rules on the abstract syntax of $Diff_{12}$ automatically updates its concrete syntax. Therefore, the final difference model is provided to the DSL user in a representation tailored for the domain.

Figure 10 illustrates the final difference model provided by our approach. It shows the application of two rules, identifying that Pacman has moved right and eaten food. Altogether, compared to Figure 3, the DSL user can inspect the domain-specific changes in an editor that

**Algorithm 3** Transformation from `SDRule` to `HRule`

```
 1: procedure GENERATEHRULE(SDRule)
 2:   HRule ← createHenshinRule(SDRule)
 3:   for all Condition c in SDRule.getConditions() do
 4:     for all Attribute a in c.getAttributes() do
 5:       p ← createHenshinParameter(a)
 6:       HRule.Parameters ← p
 7:       c.replaceAttributeByParameter(p)
 8:     end for
 9:   end for
10:   for all Pattern P in SDRule.getPatterns() do
11:     n ← createHenshinNode(P)
12:     if not P.hasFilter() AND not P.isMemberOfNACGroup() then
13:       n.Action ← "preserve"
14:     else if P.hasFilter() then
15:       n.Action ← "delete"
16:     else if P.memberOfNACGroup() then
17:       n.Action ← "forbid"
18:       n.forbidId ← P.getNACGroupName()
19:     else if P.className() == "SemanticDiff" then
20:       n.Action ← "create"
21:     end if
22:     for all Attribute a in P.getAttributes() do
23:       hAttr ← createHenshinAttribute(a)
24:       hAttr.Value ← a.getValue()
25:       n.Attributes ← hAttr
26:     end for
27:   end for
28:   for all Node n in HRule.getNodes() do
29:     P ← SDRule.getObject(n.getName())
30:     for all Association asc in P.getAssociations() do
31:       edge ← createHenshinEdge(asc.getName())
32:       edge.Source ← n
33:       edge.Target ← HRule.getNode(asc.getTarget()
34:                                   .getName())
35:       if edge.Source.getName() == "SemanticDiff" OR edge.Target.getName() == "SemanticDiff" then
36:         edge.Action ← "create"
37:       else if edge.Source.getAction() == "delete" OR edge.Target.getAction() == "delete" then
38:         edge.Action ← "delete"
39:       else if edge.Source.getAction() == "forbid" OR edge.Target.getAction() == "forbid" then
40:         edge.Action ← "forbid"
41:       else
42:         edge.Action ← "preserve"
43:       end if
44:     end for
45:   end for
46:   return HRule
47: end procedure
```

resembles the one she used to manipulate the original models *M1* and *M2*.

## 5 Conflicting rule application

A rule may have more than one match in $Diff_{12}$. However, care should be taken since applying a rule may remove filtered elements. In general, there is normally more than one semantic differencing rule specified for a DSL and different rules may have overlapping matches. In Section 4.3, the control flow of the transformation assumed the rules are sequentially independent [16]. However, if a rule filters an element that is required in the precondition of another rule, the latter will not find a match. One solution to avoid conflicts between rules is to use NACs. For example, we can prevent the application of a rule if another rule has been applied before. This can be achieved by adding a `SemanticDiff` object in the former rule as a NAC (see Section 4.1). However, this solution is limited because it alters the semantics of the rule, may prevent non-conflicting rules from applying, and requires modifying the semantic rule manually. Therefore, we propose a general solution that reduces conflicts between rules as much as possible.

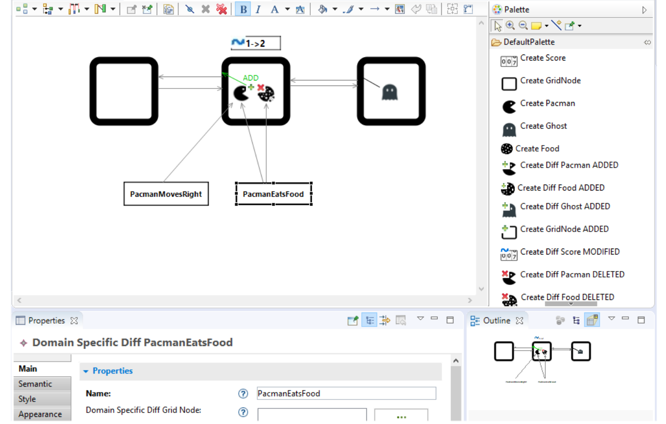


Figure 10: The domain-specific difference of two models in the generated editor after applying two rules

The problem is that multiple semantic difference rules may be applicable at the same time, and they might conflict with each other. Therefore, we extend DSMCompare with an elaborate graph-based analysis of the rules based on heuristics to obtain a reasonable schedule of the rule application order. In this case, the ordering must be such that it reduces the verbosity of the presented difference, to favor semantic differences over syntactic differences. In the following, Section 5.1 introduces an example to illustrate the conflicts that can arise, Section 5.2 formalizes the problem, and Section 5.3 proposes an algorithm to assign rules a priority.

### *5.1 Conflicting rules example*

Assume the engineer of the Pacman game DSL has defined the semantic differencing rules for the four cardinal movements of Pacman as shown in Figure 11 (a)–(d). Note that we have slightly altered the rules for illustrative purposes. After a while, some DSL users report that DSMCompare fails to detect other kinds of movements, such as diagonally or further than one grid node away. Thus, the DSL engineer creates a new rule called `Move` as depicted in Figure 11 (e). This semantic differencing

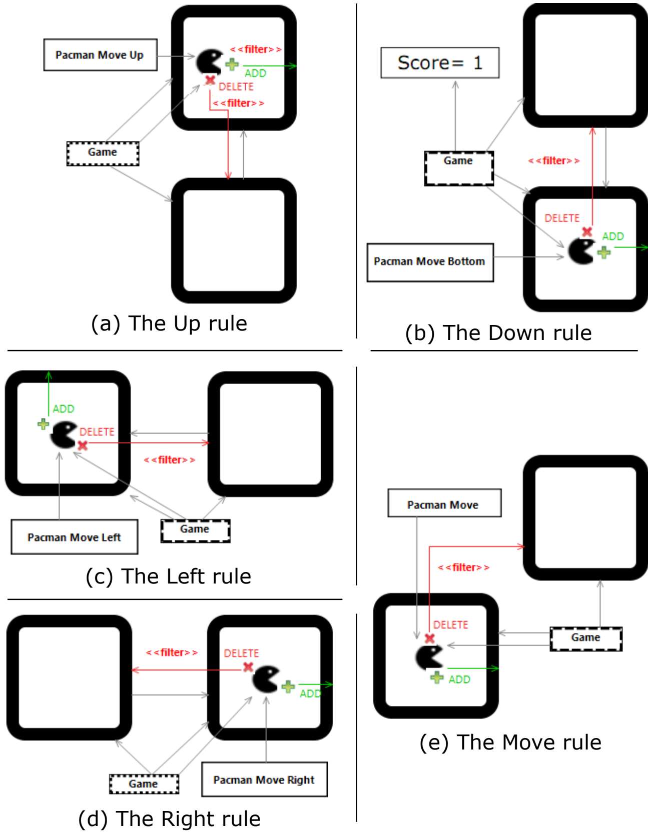


Figure 11: Semantic differencing rules for Pacman movement

rule correctly detects any change in Pacman movements. However, later, a DSL user discovers that, for some difference models, DSMCompare reports `Move` instead of the more precise `Right`. This new problem arises because the two rules conflict with each other (when `Move` is applied before `Right`): the former rule filters the old `diff_on` relation of Pacman which is required to apply the latter rule. Another situation occurs when `Move` and `Up` are both applicable, but the former is applied. In this case, the resulting $Diff_{12}$ model will contain more fine-grained differences than if the latter was applied (because `Move` filters one association, while `Up` filters two), thus encumbering the DSL user with unnecessary differences reported. This problem is further aggravated when rules have many occurrences in $Diff_{12}$. This example illustrates that, when a number of rules are in conflict, the DSL engineer should prioritize those that are more precise, remove more fine-grained differences, and create more domain-specific differences.

The DSL engineer can assign a priority to each rule thanks to their `priority` attribute (see line 16 in Algorithm 2). Priorities define a partial ordering of rule application: the lower the priority value, the higher priority the rule has. In Henshin, this is represented with a *priority unit*; thus we define the control flow of the rules with this unit instead of the independent unit presented in Section 4.3. To assist the DSL engineer in assigning the optimal priority ordering of the rules, we have developed a DSL-agnostic algorithm that proposes the best rule ordering without knowledge of the difference model $Diff_{12}$ on which they will be applied.

### 5.2 Formalization of the problem

We consider assigning priorities to the rules as an optimization problem where the objective is to maximize the number of semantic differences and minimize the number of fine-grained differences in $Diff_{12}$ after applying the rules. Intuitively, we can achieve this objective by applying as many rules as possible. However, some rules may filter more fine-grained differences than others and some rules may create more semantic difference objects than others. The latter may seem unusual because, typically, one rule creates a single semantic difference object that represents the intention of the rule. However, our framework allows the DSL engineer to define higher-order semantic differencing rules that refactor semantic difference objects created by other rules.

Therefore, the solution should consider conflicts between the rules, the number of filtered elements they remove, the number of semantic difference objects they create, and the number of overlaps between them to favor more precise rules (like `Right`) over less precise ones (like `Move`). We represent this information in a *conflict graph* where vertices are rules and edges represent conflicts between them. The priority assignment solution comes down to sorting every vertex of the graph while optimizing our objective.

#### 5.2.1 Conflict graph

We define the conflict graph as $G = \langle V, E, sem, filter, elem, conf \rangle$ with $sem, filter, elem : V \to \mathbb{N}$ properties of vertices, $E \subseteq V \times V$ irreflexive directed edges, and $conf : E \to \mathbb{N}$ the weights of edges.

In this representation, each vertex $v \in V$ corresponds to a rule. Vertices have the following properties:

- *sem* is the number of semantic difference objects each match of the rule will create on $Diff_{12}$.
- *filter* is the number of fine-grained differences each match of the rule will filter.
- *elem* is the number of class and association instances to be matched by the pattern of the rule.

The vertices of the conflict graph in Figure 12 show the properties of each rule of the Pacman game presented in Figure 11. An edge $(v_1, v_2) \in E$ represents a conflict that occurs if we apply the rule corresponding to $v_1$ before the rule corresponding to $v_2$. Since we assume that a rule is applied on all matches exhaustively before applying another one, edges cannot be reflexive.

Edges are weighted by function *conf*, which gives the number of conflicts that arise when applying the rule of the source vertex before the rule of the target vertex. Following the theory of graph transformation with NACs [33], we consider two kinds of conflicts for rules:

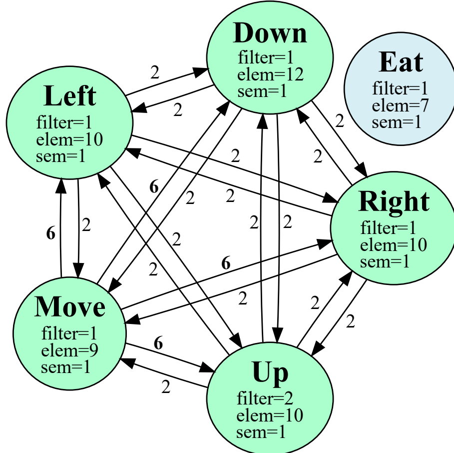


Figure 12: The conflict graph for the rules in Figure 11

– *Delete-use* occurring when a rule deletes an element (e.g., a fine-grained diff) that another rule requires. An example of this conflict is when `Move` filters an association required by `Up`.
– *Produce-forbid* occurring when a rule creates an element that another rule forbids in a NAC. An example of this conflict would be when a rule creates a semantic diff that another rule forbids.

Finding an optimal solution to the problem is equivalent to finding an optimal vertex partial ordering according to our objective. The solution is a function $priority : V \rightarrow \mathbb{N}$ such that if $priority(v_1) < priority(v_2)$, then DSMCompare should try to apply the rule corresponding to $v_1$ before the rule corresponding to $v_2$. If $priority(v_1) = priority(v_2)$ then the rules are not in conflict and can be applied in any order.

*5.2.2 Conflict detection* To compute the edges of the conflict graph and their weight, we perform a conflict analysis of the rules. Henshin offers a multi-granular conflict and dependency analysis tool (MultiCDA), a generalization of critical pair analysis (CPA) [34]. Conflicts need to be detected only once by the DSL engineer, thus the computation time of conflicts is not an issue for our problem. Nevertheless, MultiCDA is significantly faster than CPA [34]. Given a set of Henshin rules, MultiCDA outputs three levels of conflict granularity. To assign the $conf$ weight to each edge of the conflict graph, we rely on the fine-granularity level that MultiCDA reports. It outputs a positive integer for each pair of rules representing the number of all model fragments whose presence leads to a conflict. MultiCDA presents the conflict results as a matrix. This serves as the adjacency matrix of our conflict graph (note that we assign 0 to the main diagonal since edges are irreflexive).

Applying conflict detection with MultiCDA on the Pacman game semantic differencing rules in Figure 11 results in the edges of the graph in Figure 12. The `Eat` rule has no conflicting model fragment with any other rules, thus it is disconnected. The edges outgoing from `Move` indicate that if we apply this rule before any of the other movement rules, there are six model fragments that lead to conflicts. In contrast, applying any of the cardinal movement rules before any other movement rule causes conflicts only for two model fragments. For example, one of them is: `[Pacman]--(diffon)--[DiffPositionableEntity_on]--(eType)--[GridNode]`. Applying the `Move` rule on this model fragment will remove the three central elements, whereas all the other movement rules require this fragment to be applicable.

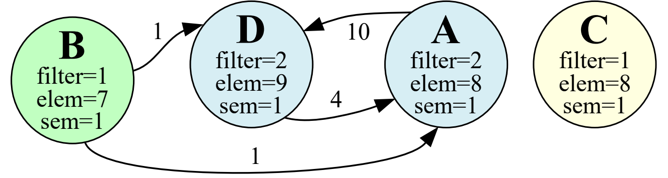


Figure 13: A sample conflict graph

### 5.3 Rule priority ordering

To illustrate how to solve the rule priority ordering, consider the conflict graph in Figure 13. It represents the conflicts between four semantic differencing rules $A, B, C$, and $D$ encoded by vertices with the same name. Intuitively, a solution to the problem is to sort the vertices of the conflict graph topologically. However, recall that the edge $(B, D)$ means that when $B$ is applied on a model fragment, $D$ is no longer applicable on this fragment. Therefore, we must consider reversing the edges before the topological sort. However, topological sorting algorithms are only applicable to directed acyclic graphs. Since conflict graphs are likely to contain cycles and vertices are weighted, only approximate algorithms exist in the literature [2]. Nonetheless, our goal is to assign a partial order to all vertices such that applying a rule with lower order will less likely prevent the application of other rules while maximizing $filter, elem$, and $sem$. Therefore, we propose an algorithm (Algorithm 4) that sorts weighted vertices and edges of a directed cyclic graph based on heuristics.

Algorithm 4 starts by partitioning the conflict graph $G$ into two disjoint subgraphs. The left graph $L$ contains the maximum subgraph of $G$ that is acyclic. The right graph $R$ is the graph induced by the remaining vertices.

ToDag() transforms a graph into a directed acyclic graph by iteratively removing vertices from the strongly connected components. We implement Tarjan's algorithm [63] to find the strongly connected components of the graph in $O(|V|+|E|)$ time complexity. If $G$ is the conflict graph in Figure 13, then the strongly connected components are the subgraphs $\langle A, D\rangle$, $\langle B\rangle$, and $\langle C\rangle$. Thus, to make $L$ acyclic, we should remove either vertex $A$ or $D$. We define the following heuristics (in this order), to choose which vertex to remove from a strongly connected

**Algorithm 4** Priority ordering of the vertices of a conflict graph

```
1: procedure PriorityOrder(G)
2:   L ← G.clone()
3:   R ← G.clone()
4:   L ← ToDag(L)
5:   sortedL ← RevTopologicalSort(L)
6:   R ← R − L
7:   if not IsDAG(R) then
8:     sortedR ← PriorityOrder(R)
9:   else
10:    sortedR ← RevTopologicalSort(R)
11:  end if
12:  sort ← sortedL + sortedR
13:  priority(v) ← 1, ∀v ∈ sort
14:  for all v in sort do
15:    before ← {u | (u, v) ∈ E ∨ (v, u) ∈ E
16:                    and u is before v in sort}
17:    if |before| > 0 then
18:      priority(v) ← max{priority(u), ∀u ∈ before} + 1
19:    end if
20:  end for
21:  return priority
22: end procedure
```

component (we denote its set of vertices by $S$) until $L$ has no more cycles:

$H_1 = \max \sum_{v \in S} \sum_{(v,u) \in E} conf(v, u)$ maximizes the total weight of the outgoing edges of a vertex $v$, to choose the rule with the highest number of fine-grained conflicts.

$H_2 = \max \sum_{v \in S} |\{(u, w) \in E \mid v = u \vee v = w\}|$ maximizes the degree of a vertex $v$, to choose the rule with the highest number of conflicting rules.

$H_3 = \min \sum_{v \in S} sem(v)$ serves to choose the rule that creates the least number of semantic difference objects.

$H_4 = \min \sum_{v \in S} filter(v)$ serves to choose the rule that filters the least number of granular difference objects.

$H_5 = \min \sum_{v \in S} elem(v)$ serves to choose the rule that matches the lowest number of elements in the difference model, thus the least precise rule.

Hence, $L$ contains the vertices representing rules that are less likely to prevent the application of other rules and optimize our objective. In the conflict graph of Figure 13, heuristic $H_1$ suffices to remove $A$ from $L$. All vertices of $L$ will be given a lower priority value than vertices of $R$. Thus, it is important that we minimize the size of $R$. In our example, $R$ consists only of vertex $A$. Since $L$ is now acyclic, we apply RevTopologicalSort() to sort the vertices of $L$ in reverse order of the edges using a $O(|V| + |E|)$ time complexity algorithm based on depth-first search. During the traversal, we use the opposite of the five heuristics whenever we have a choice between more than one vertex (i.e., we minimize $H_1, H_2, H_5$ and maximize $H_3, H_4$).

On line 12, *sort* contains the sequence of vertices sorted topologically. In our example, $sort = (D, B, C, A)$ the first three from *sortedL* and the last one from *sortedR*. The algorithm constructs the *priority* function by following the order of the vertices in *sort*. However, this total order is overly conservative, e.g., $C$ has no conflict with the other rules. On lines 15–18, we ensure that if $u$ is topologically before $v$ and there is an edge between $v$ and $u$, then $priority(v) > priority(u)$. Otherwise, they can have the same order. The *priority* function output for the conflict graph in Figure 13 is presented in Table 1. The table also shows the initial value of the heuristics of each vertex.

Table 1: Priority order of the sample conflict graph in Figure 13 output by the algorithm

| **Rule** | **Priority** | $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ |
|---|---|---|---|---|---|---|
| C | 1 | 10 | 3 | 1 | 2 | 8 |
| D | 1 | 2 | 2 | 1 | 1 | 7 |
| B | 2 | 0 | 0 | 1 | 1 | 8 |
| A | 3 | 4 | 3 | 1 | 2 | 9 |

When removing vertices from $L$ to make it acyclic, we may end up with an induced graph $R$ with the removed vertices that still contains cycles. For example, this happens if $G$ is a complete graph, then $L$ can only consist of one vertex that optimizes the heuristics. This is the case with the conflict graph of the Pacman game example in Figure 12. Since its vertex is disconnected, `Eat` can be applied first and be part of $L$. All the rest of the vertices are in a clique, thus applying one would conflict with all others. However, we want to give as much chance as possible to apply as many rules as possible to optimize $H_3$. Nevertheless, only one of the movement rules can remain in $L$. According to $H_1$, `Move` has the highest number of conflict reasons, so it should be applied last and be part of $R$. All the other four vertices have the same $conf$ value. According to $H_4$, `Up` should have the lowest priority value among them and be part of $L$. Thus, all remaining rules are part of $R$, still forming a clique. Therefore, on line 8, we recursively order $R$ until it is acyclic. Rules `Left`, `Right`, and `Down` are structurally very similar, except the latter which has one more element (the scoreboard). Semantically, this means that `Down` is more precise than the other two rules because it requires matching more elements. Applying another rule may risk removing this additional element, and thus not allowing `Down` to be applicable anymore. Therefore, according to $H_5$, `Down` should have a lower priority value than the other two rules. `Left` and `Right` rule cannot be further distinguished. Hence, any order between them will lead to the same chance of making the other inapplicable. Table 2 summarizes the order generated by Algorithm 4. The table also shows the initial value of the heuristics of each vertex.

Table 2: Priority order of the Pacman game rules output by the algorithm

| **Rule** | **Priority** | $H_1$ | $H_2$ | $H_3$ | $H_4$ | $H_5$ |
|---|---|---|---|---|---|---|
| Eat | 1 | 0 | 0 | 1 | 1 | 7 |
| Up | 1 | 8 | 8 | 1 | 2 | 10 |
| Down | 2 | 8 | 8 | 1 | 1 | 12 |
| Right | 3 | 8 | 8 | 1 | 1 | 10 |
| Left | 4 | 8 | 8 | 1 | 1 | 10 |
| Move | 5 | 24 | 8 | 1 | 1 | 9 |

Since our objective depends on the $Diff_{12}$ model, but the conflict graph is agnostic from any model (i.e., it only depends on the rules), the priority order output may not be optimal for all $Diff_{12}$ models. Nevertheless, it should be optimal for most models. If the conflict graph contains no cycle, applying the rules in the order output by Algorithm 4 essentially allows all rules to apply on any input models without conflict. However, if there are cycles, the order output does not prevent conflicts but minimizes their impact. Thus, this increases the probability of replacing a maximum number of fine-grained differences with semantic differences.

### *5.4 Extensions*

Some extensions to the heuristics we present could be considered. In particular, the goal of $H_5$ is to favor more precise rules as a last resort. Currently, *elem* only counts the elements to be matched in a rule. One could argue that a rule with NACs is more precise than one without, since it has fewer chances of matching. Thus it could be possible to count NAC elements in *elem*. One could also argue that a rule with abstract elements is less precise than a similar rule using one of its subclasses. For example, consider the `Move` rule in Figure 11 (e). Suppose we had another rule `MoveAny` that relied on the `PositionableEntity` class instead of the `Pacman` class. Then `Move` can be considered more precise than `MoveAny`, since it has fewer chances of matching. Therefore *elem* could take into consideration abstract classes and inheritance relations.

## 6 Evaluation

Next, we evaluate DSMCompare using both synthetic models (Section 6.3) and model histories created by third parties (Section 6.4). We first briefly outline the implementation of DSMCompare. Then we state the objectives of our evaluation in Section 6.2. We present the two sets of experiments (Sections 6.3 and 6.4), discuss the results in Section 6.5 and present limitations and threats to validity in Section 6.6.

### *6.1 Implementation*

We implemented DSMCompare as an Eclipse plug-in running on the Eclipse Modeling Framework (Eclipse version 2020-09). It is available on the companion website[5].

Given a DSL, DSMCompare automatically generates out of the box all required artefacts to support the visualization of model differences for the DSL (i.e., diff metamodel, fine-grained diffs, and extended concrete syntax). Then, if so desired, the DSL designer can provide domain-specific semantic diff rules, as these rules cannot be inferred automatically. If the DSL evolves, the DSL designer would have to evolve the semantic diff rules as well, but the rest of artefacts can be regenerated again with no effort.

To perform the model comparison, DSMCompare consists of three main modules. The *Comparison* module takes as input two model versions and produces the corresponding fine-grained $Diff_{12}$ model. This module relies on the EMF-Compare model comparison tool (version 3.3.9). The *Ordering* module computes the priority order of the SDRules to be applied. It first transforms the SDRules into Henshin rules. Then, it invokes the Henshin's Multi-CDA tool (version 1.7) to retrieve the potential conflicts among the rules. The ordering module takes the conflicts and the SDRules to produce the scheduling units of the Henshin transformation. Finally, the *Lifting* module applies this transformation on the $Diff_{12}$ model to obtain the semantically lifted $Diff_{12}$ model. The difference model is then fed to generated Sirius editor (version 6.3.0) to present the semantic $Diff_{12}$ model in concrete syntax.

To use DSMCompare for a given DSL, the DSL Engineer needs to perform two manual tasks. The first one is to assign an appropriate concrete syntax representation to the classes and relationships generated in the DSDiff metamodel. The engineer only needs to consider the elements prefixed with "Diff". For each Diff class, she needs to create three versions (ADD, DELETE, MODIFY) of the concrete syntax for the diff class corresponding to the original metamodel of the DSL. For example, as depicted in Figure 10, we created three additional icons representing Pacman by adding a $+/\times/\sim$ symbol respectively. Similarly, the engineer needs to create two versions (ADD, DELETE) of the concrete syntax for the diff association corresponding in the original metamodel of the DSL. The second task is to create the SDRules for the DSL. The number of SDRules to create depends on the DSL; for example, Pacman required 12 rules, Arduino (cf. Section 6.4.1) 24 rules, and Class Diagram Refactoring (cf. Section 6.4.2) 20 rules. In general, writing a SDRule is advantageous over writing the equivalent Henshin rule. The generated domain-specific editor (e.g., in Figure 8) and the abstraction level that deals directly with concepts of the DSL reduce the effort compared to creating Henshin rules using generic nodes and edges, and adding

[5] `https://github.com/geodes-sms/DSMCompare/`

explicitly graph transformation inscriptions (e.g., NAC groups as shown in Algorithm 3).

### 6.2 Objectives

Our first goal is to evaluate if DSMCompare improves the readability and understandability of differences between model versions. To this end, we characterize the verbosity of the differences formulated by two research questions:

**RQ1** Are fine-grained differences more verbose than semantic differences?
**RQ2** Does assigning priorities to semantic differencing rules yield less verbose difference models?

We claim that the more differences are presented to a domain user, the harder it is for her to comprehend the changes that differentiate two models from a semantic point of view. Therefore, RQ1 investigates whether presenting more semantic differences rather than fine-grained differences, reduces the verbosity of the difference model. RQ2 focuses on the impact of the priority ordering of the semantic differencing rules in decreasing the verbosity. The metrics we use to answer both research questions are the number of remaining fine-grained differences and the number of discovered semantic differences in the difference model. To answer RQ2, we use synthetic models from two scenarios (the Pacman game and metamodel refactorings) as we will detail in Section 6.3.

The second goal is to evaluate the applicability of our approach in finding semantic differences between model versions. We concentrate on the following two research questions:

**RQ3** Can we extract semantic differences from fine-grained diffs?
**RQ4** Are semantic differences recurring?

RQ3 assesses whether semantic differencing rules are applicable in practice. However, these rules must be applicable to any difference model of a particular DSL. If a rule is rarely applicable on a set of models, then the rule is too specific to certain classes of models of the DSL and general enough to the DSL. Therefore, we must ensure that semantic differencing rules are recurring. The metric we use to answer these latter two research questions is the number of occurrences of semantic differences over model histories created by third parties, as we will detail in Section 6.4.

### 6.3 Reducing the verbosity with semantic differencing

We present the first experiment to evaluate if DSMCompare yields less verbose difference models.

#### 6.3.1 Experimental setting

*Cases.* In this experiment, we consider two cases: the Pacman game configuration DSL (Pac-Man) presented in previous sections, and the refactoring of Ecore metamodels (MM-Refactoring). We choose these two cases to vary the size of the difference models, the number of semantic differencing rules, and the topology of the conflict graph. Moreover, the reasons for the selection of the second case are twofold. On the one hand, it illustrates that our approach works for both models and metamodels, by just looking at Ecore metamodels as instances of (i.e., models of) the Ecore meta-metamodel. On the other hand, GitHub contains many Ecore metamodels, which increases the chances of finding interesting metamodel version histories for our experiment.

For the Pac-Man case, we have specified 12 semantic diff rules: five for Pac-Man movements (up, down, left, right, and the general move), five similar rules for ghost movements, one for Pac-Man eating food, and one for a ghost killing Pac-Man. Every rule has one filter and creates one semantic difference object. The conflict graph of the rules forms three disconnected cliques: one for ghost movements, one for Pac-Man movements with the Pacman-Die rule, and the disconnected Pacman-Eat rule. All rules are composed of eight elements, except the Pacman-Die rule which is composed of seven. The Pac-Man case represents situations where the semantic difference rules are uniform.

For the MM-Refactoring case, we have specified 20 semantic difference rules adapted from the metamodel and object-oriented refactoring catalogs[6], such as Extract-Superclass, Split-References, and Rename-Attribute. The conflict graph of the rules forms two disconnected graphs. The first graph contains four rules, three of them (for method movement) forming a strongly connected component. The second graph comprises strongly connected components of 13 rules: eight for references and five for attributes. All rules filter one or two elements, except the three renaming rules, which have no filter. They are all composed of five to nine elements. As opposed to the Pac-Man case, the MM-Refactoring case represents situations where there is more variability between the semantic differencing rules.

For both cases, we used DSMCompare to generate the corresponding *DSDiffMM* and *SDRuleMM* metamodels. We specified the semantic differencing rules with the generated editor and automatically transformed them into Henshin to apply them on a set of difference models. All the material such as models, data, rules and conflict graphs are available on the companion website.

*Model generation.* To address RQ2, we want to verify that applying DSMCompare to a difference model maximizes the number of semantic difference objects and minimizes the number of fine-grained differences. Since

[6] `https://www.metamodelrefactoring.org` and `https://refactoring.com` respectively

Table 3: Results of applying the semantic differencing rules in different orders on the difference models. The numbers in the form $x \mid y$ represent $x$ semantic difference objects and $y$ fine-grained differences remaining in the difference model after applying all semantic differencing rules in the corresponding order.

<table>
<tr><th>DSL</th><th>Diff model</th><th>#fine-diffs</th><th>Without conflicts</th><th>Ordered</th><th>Reverse order</th><th>Random order 1</th><th>Random order 2</th><th>Random order 3</th><th>Random order 4</th><th>Random order 5</th><th>Random order 6</th><th>Random order 7</th></tr>
<tr><td rowspan="5">Pac-Man</td><td>M1</td><td>90</td><td>77 | 0</td><td>60 | 30</td><td>28 | 45</td><td>34 | 40</td><td>45 | 29</td><td>55 | 42</td><td>60 | 23</td><td></td><td></td><td></td></tr>
<tr><td>M2</td><td>52</td><td>42 | 0</td><td>28 | 24</td><td>22 | 15</td><td>23 | 15</td><td>24 | 15</td><td>28 | 20</td><td>28 | 16</td><td></td><td></td><td></td></tr>
<tr><td>M3</td><td>49</td><td>41 | 0</td><td>32 | 17</td><td>16 | 24</td><td>16 | 24</td><td>27 | 14</td><td>27 | 15</td><td>32 | 17</td><td>–</td><td>–</td><td>–</td></tr>
<tr><td>M4</td><td>68</td><td>67 | 0</td><td>44 | 24</td><td>23 | 29</td><td>28 | 24</td><td>38 | 16</td><td>39 | 17</td><td>44 | 19</td><td></td><td></td><td></td></tr>
<tr><td>M5</td><td>62</td><td>46 | 0</td><td>32 | 30</td><td>16 | 31</td><td>16 | 31</td><td>24 | 24</td><td>29 | 27</td><td>32 | 30</td><td></td><td></td><td></td></tr>
<tr><td rowspan="5">MM-Refactoring</td><td>M1</td><td>337</td><td>219 | 90</td><td>117 | 228</td><td>92 | 230</td><td>111 | 229</td><td>100 | 226</td><td>99 | 235</td><td>95 | 227</td><td>117 | 234</td><td>117 | 228</td><td>117 | 234</td></tr>
<tr><td>M2</td><td>262</td><td>88 | 183</td><td>57 | 223</td><td>53 | 217</td><td>54 | 223</td><td>55 | 222</td><td>55 | 223</td><td>53 | 219</td><td>57 | 223</td><td>57 | 222</td><td>57 | 223</td></tr>
<tr><td>M3</td><td>266</td><td>88 | 188</td><td>71 | 188</td><td>66 | 210</td><td>69 | 212</td><td>66 | 210</td><td>69 | 213</td><td>66 | 211</td><td>71 | 213</td><td>71 | 213</td><td>71 | 212</td></tr>
<tr><td>M4</td><td>248</td><td>65 | 175</td><td>53 | 195</td><td>48 | 192</td><td>51 | 194</td><td>48 | 192</td><td>48 | 193</td><td>48 | 192</td><td>48 | 193</td><td>53 | 195</td><td>53 | 194</td></tr>
<tr><td>M5</td><td>277</td><td>139 | 123</td><td>79 | 197</td><td>71 | 195</td><td>73 | 195</td><td>73 | 191</td><td>72 | 200</td><td>71 | 197</td><td>79 | 200</td><td>79 | 194</td><td>79 | 200</td></tr>
</table>

it is not tractable to test exhaustively all possible difference models of each DSL, we derive a representative set of difference models covering most cases. Therefore, we construct five difference models (M1 to M5) by varying the number of occurrences of each rule when applied in isolation, i.e., assuming there are no conflicts between rules.

We constructed M1 by hand, ensuring that all semantic differencing rules have an almost equal number of matches when applied in isolation (an average of $6 \pm 1$ matches for Pac-Man and $10 \pm 3$ for MM-Refactoring). Therefore, M1 represents models where the number of matches of each rule is uniformly distributed, regardless of any priority order. For the remaining models, we randomly varied the skewness and kurtosis of the number of matches of each rule depending on their priority order output by Algorithm 4.

In M2 of the MM-Refactoring, we favor the number of matches of the 10 highest and lowest priority rules to cover 90% of all the matches. Similarly for Pac-Man, we favor the number of matches of the 6 highest and lowest priority rules covering 84% of all the matches. For example, the Pacman-Eat (top priority) and Pacman-Move (lowest priority) rules have six and eight matches, whereas Ghost-Left has only one match. Since lower priority rules have many conflicts with higher ones, M2 represents difference models where the priority ordering is least optimal: the lower priority rules will likely not be applicable.

In M3, we favor the 6 and 10 highest priority rules for Pac-Man and MM-Refactoring respectively. All remaining rules have at most one match. Therefore, M3 represents difference models where the priority ordering is optimal.

In M4, we favor the same number of lowest priority rules as in M3, while all higher priority rules have at most one match. For example, in the MM-Refactoring, the Merge-Reference rule (top priority) has no match, whereas Remove-Middle-Man (lowest priority) has five matches.

Finally, in M5, the highest and lowest priority rules have at most one match while favoring the matches of all other rules.

The first four columns of Table 3 summarize the setup of each case. The *#fine-diffs* column shows the total number of fine-grained differences in each difference model before applying the semantic differencing rules. For instance, there are 90 fine-grained differences for M1 of the Pac-Man DSL, among which 76 are association differences and 14 are class differences. To better characterize each model, the next column (labelled *Without conflicts*) shows the total number of matches[7] of all semantic differencing rules when run in isolation, assuming there are no conflicts between the rules. This gives an idea of how many times the rules are applicable; though this number is not reachable when there are conflicts between rules. For example, for M1 of the Pac-Man DSL, if all rules were to be applied on all their matches, the resulting difference model would contain 77 semantic differences, and all of the 90 fine-grained differences would be filtered. Note that the difference models for MM-Refactoring are on average 4.5 times larger than those for Pac-Man.

*Priority orderings.* The first independent variable of this experiment is the difference model (M1–M5) to avoid a bias in the priority order output by our approach. Furthermore, to answer RQ2, we must compare the order output by DSMCompare with other orders. One interesting order we can compare with is the reverse order. This allows the rules with most conflicts to be applied first. Other orders to compare to are obtained through random sampling from all possible permutations. However, it is intractable to test against all possible permutation of rule ordering. One property of Algorithm 4 is that rules with the same priority have no conflict between them. We denote rules sharing the same priority as a *cluster*. Thus, the order within each cluster does not have an impact on the other rules. Therefore, we can ignore the permutations within clusters. For the Pac-Man case, we obtain 6 clusters for the 12 rules and for the MM-Refactoring case, we obtain 9 clusters for the 20 rules. Still, manually testing all these possible permutations is not feasible (720 and 362 880 for the Pac-Man and MM-Refactoring cases

[7] Since each rule creates a single semantic difference object, this number is the same as the number of matches.

respectively). In the random sampling, we generated orders such that no cluster has the same priority twice. Therefore, there are as many orders as there are clusters. After excluding the order output by Algorithm 4 and the reverser order, we end up with 4 additional random orders for Pac-Man and 7 for MM-Refactoring cases.

*6.3.2 Results* Table 3 shows the results of this experiment. In bold, we highlighted the cases where the metrics are optimized: maximizing the number of semantic differences and minimizing the number of fine-grained differences. For example, for the M1 model of Pac-Man, applying the rules in a priority ordering output by DSMCompare results in 60 semantic differences with 30 fine-grained differences remaining. In all the tested cases, the results show that the priority order output by Algorithm 4 maximizes the number of semantic differences. Nevertheless, for Pac-Man, one of the random orderings filters more fine-grained differences than our order in three difference models. After manual inspection, we identified that this is because, in this ordering, the Pacman-Move rule has a higher priority than Pacman-Right. However, this means that a more general semantic difference takes precedence over a more specific semantic difference. This contradicts our heuristic $H_5$, which favors the latter over the former. This is a desirable property of our ordering since, in practice, if the Pac-Man moved to the right, then we would like that the difference model depicts the direction in which it moved.

For MM-Refactoring, our priority order produces the best results in terms of the metrics collected. We notice that two random orders obtain slightly fewer fine-grained differences. Like for Pac-Man, they also give lower priority order to more general rules, such as Move-Reference. However, since they filter more fine-grained differences than more specialized rules, the same number of fine-grained differences are filtered overall.

Regarding RQ1, we can conclude that the fine-grained differences are more verbose since semantic differences aggregate multiple fine-grained differences. Regarding RQ2, we find that assigning priorities has a significant influence on the verbosity of the difference model. Furthermore, we notice that most of the time, our ordering results in less verbose difference models. Although it does not always optimize the number of fine-grained differences, it reports more precise semantic differences. We believe maximizing this aspect improves the readability of the model on top of reducing the number of fine-grained differences.

### *6.4 Case studies*

We now validate our approach on two real-life case studies developed by third-parties. The first case we choose is a DSL with a graphical concrete syntax and a few model versions on which we apply DSMCompare. In the second case, we focus on larger models with many versions available.

#### *6.4.1 Arduino Designer*

*Description.* Arduino Designer is an environment specially tailored to young children, to create simple programs for Arduino[8], an open-source electronics platform based on easy-to-use hardware and software. The Arduino Designer language is a DSL built to model Arduino configurations and programs graphically, based on Sirius. The DSL has two parts: one for the configuration of devices and another for sketching programs. The configuration part contains primitives for placing hardware devices on the appropriate pins of the Arduino board. In Arduino, the code is placed and executed within a main loop. The sketch part models the code within the loop. It is a graphical programming language with arithmetic expressions, loops, and conditional instructions.

Just like code, these models evolve in new versions. For example, in a GitHub repository[9], we can find a history of different models that underwent bug fixes, improvements, and migrations to a new framework. Understanding complex changes that have occurred from one version to another may be hard for Arduino developers, especially if they are children. Our approach can help these developers visualize the changes in the same graphical language and environment they used for development. Furthermore, we report the changes as semantic differences. For the sketch part, we reuse known code refactoring patterns and model them as semantic differencing rules. The changes in the configuration part typically consist of adding or replacing devices in appropriate pins of the board.

*Domain-specific comparison of Arduino models.* We have applied DSMCompare on different versions of Arduino models available in the repository. The original metamodel *ArduinoMM* consists of 36 classes, 33 associations, and 17 attributes. The concrete syntax *ArduinoCS* assigns an icon for every class and association. With DSMCompare, we generate the difference metamodel *ArduinoDiffMM* with 96 classes, 137 associations, and 110 attributes. The rule metamodel *ArduinoRuleMM* contains one more class and association, with 219 attributes. The generated concrete syntax definitions are of a similar scale.

The Arduino GitHub repository includes 13 working example projects. We filtered 6 of them, since they had an initial empty model, and just another version adding all model elements. We applied DSMCompare on all remaining 7 projects, and Table 4 summarizes the results. Each model has between 2 and 4 versions in the repository. The commit message associated with a version helped us to identify the purpose of the model changes (shown in the *Version n* and *Version n+1* columns). The fourth column (*Fine Diffs*) shows the total number of fine-grained

[8] https://www.arduino.cc/

[9] https://github.com/mbats/arduino/

Table 4: Comparison of model versions in the Arduino Designer examples repository

| Project | Version n | Version n+1 | Fine Diffs | Semantic Diff Rules | Occurrences | Remaining Fine Diffs |
|---|---|---|---|---|---|---|
| alarmlight | *Repeat* | *Fix generation for alarm light example* | 32 | Change Digital Pin<br>Change Next instruction<br>Delete a Status<br>Delete a loop<br>Replace a loop<br>Refactor a Repeat loop<br>Add a Status | 1<br>7<br>2<br>1<br>2<br>1<br>2 | 25 |
| | *Fix generation for alarm light example* | *Fix alarm light example* | 6 | Change Next instruction<br>Delete a Status | 4<br>1 | 5 |
| | *Fix alram light example* | *Migrate alarmlight example to sirius* | 31 | Change Digital Pin<br>Change an Output Module<br>Change Status<br>Change Repeat Iteration<br>Delete a Status<br>Add a Status<br>Replace a loop<br>Change Next instruction<br>Change Delay Value | 1<br>2<br>2<br>1<br>1<br>2<br>1<br>2<br>3 | 24 |
| fadelight | *While* | *Sub instructions* | 21 | Change Next instruction<br>Refactor a While loop<br>Add a Level | 1<br>2<br>2 | 11 |
| | *Sub instructions* | *Create variable, constant, math operator* | 29 | Replace a loop<br>Change Next instruction<br>Change While Condition<br>Refactor a While loop | 2<br>2<br>2<br>1 | 24 |
| | *Create variable, constant, math operator* | *Generate while* | 10 | Change Next instruction<br>Delete a loop | 1<br>1 | 6 |
| infraredsensor | *Support infrared and servo* | *Migrate infrared sensor example* | 2 | Change Digital Pin | 2 | 1 |
| | *Migrate infrared sensor example* | *Migrate examples to sirius 2.0.3* | 1 | Change Next instruction | 1 | 1 |
| servo | *Support infrared and servo* | *Migrate servo example to sirius* | 5 | Replace a connector<br>Change Next instruction | 1<br>1 | 3 |
| tigger.all | *Add Tigger example* | *Update the tigger example* | 25 | Change Next instruction<br>Refactor an If condition<br>Add a Status<br>Set Repeat condition<br>Add a Level<br>Add a Sensor | 1<br>1<br>4<br>1<br>2<br>1 | 18 |
| tigger.bubble | *add tigger bubble example* | *Fix issue on bubble example* | 2 | Change Status | 2 | 2 |
| tigger.tail | *Add Tigger tail example* | *Update tail example* | 4 | Delete an Output Module<br>Change Connector | 1<br>1 | 2 |
| | *Update tail example* | *Update cat tail example to add miaou sound* | 20 | Add an Output Module<br>Add Connerctor<br>Replace an If condition<br>Add a Status<br>Move Delay | 1<br>1<br>1<br>2<br>2 | 13 |

differences found by DSMCompare. For example, in the `fadelight` project, when comparing the version *While* and the version *Sub instructions* (versions 1 and 2 of this project), DSMCompare reported 21 fine-grained differences. The column *Semantic Diff Rules* shows the name of the semantic differencing rules recognized among the fine-grained differences, and column *Occurrences* represents the number of occurrences of each rule. Finally, the last column shows the number of remaining fine-grained differences after some differences were removed (filtered) by applying the semantic diff rules.

*Results.* Table 4 clearly shows that DSMCompare is able to extract semantic differences from fine-grained differences, being able to report one or more semantic differences across all versions of the considered projects. Moreover, most semantic diff rules (13 out of 24, 54%) where applied several times, and 29% of them were applied across different projects.

As an illustration, for the `fadelight` project, DSMCompare reported two semantic differences of type *"Refactor a while loop"*, representing a while-loop refactoring (cf. Figure 14). The first while-loop sets the device for a specific time in the *on* state, and the second loop models the *off* state of a *"FadeLight"*. In addition to one class difference, each of the two semantic diffs has also two diffs of associations. One of them represents the *"condition"* of the while-loop, and the other a link to the *"next"* instruction after the loop.

As expected, the fine-grained $Diff_{12}$ models contain fewer changes (cf. last column of Table 4) after applying the semantic diff rules. For example, the `tigger.tail` model adds an infrared sensor to a digital pin, and a servo motor to another digital pin in the Arduino board. The board also adds instructions to the end of the main loop. In this case, the fine-grained $Diff_{12}$ model shows the removal of six fine-grained differences and the addition of 14 fine-grained differences (a total of 20 changes). These changes can be encapsulated in five semantic differencing rules, i.e., *"Add an Output Module"*, *"Add Connector"*, *"Replace an If condition"*, *"Add a Status"*, and *"Move Delay"*. These rules correspond to the intention of the change i.e., *"add miaou sound to the cat"*. Meanwhile, seven fine-grained differences have been removed.

Most of the identified semantic differences in Table 4 are additions to an already designed Arduino model related to fix bugs, make improvements, or migrate to a new framework. Due to the nature of the Arduino DSL, any insertion of a device in the configuration part also requires changes in the sketch part. In `fadelight`, only the sketch part of the model is affected as we are inserting a while-loop to turn the LED light on and off gradually.

#### *6.4.2 Class Diagram Refactoring*

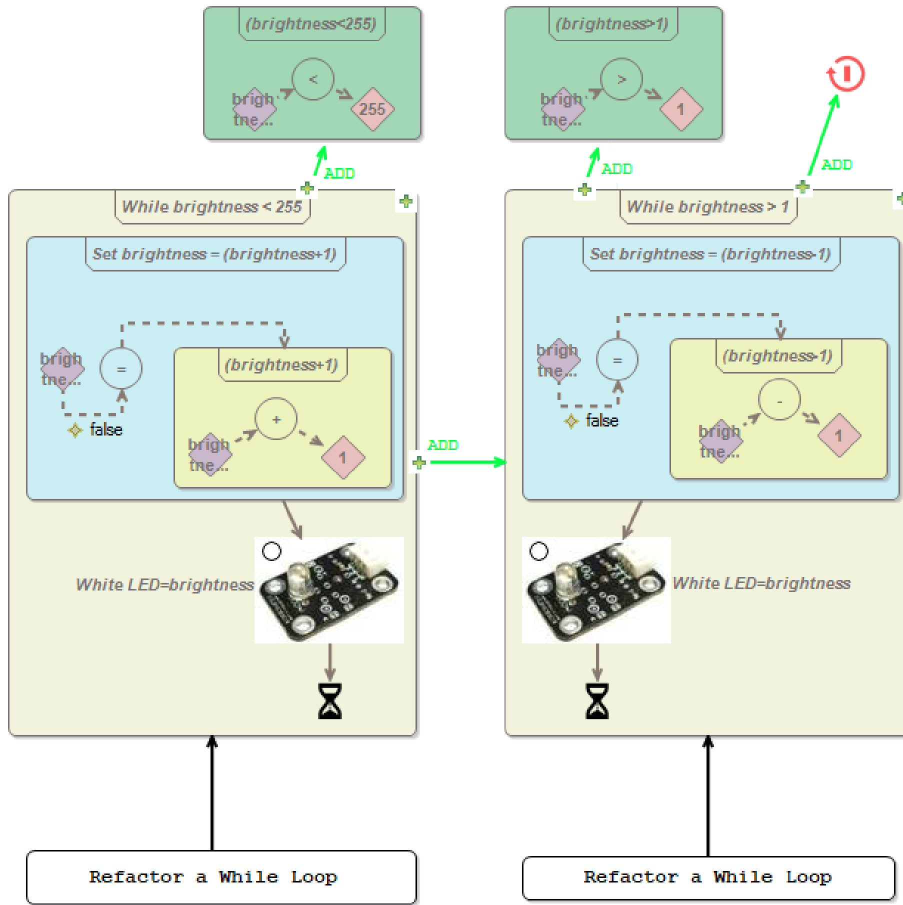


Figure 14: Domain-specific differences in Arduino designer for the `fadelight` project

*Description.* The second case study is about refactoring class diagram models. We focused particularly on refactoring metamodels defined in Ecore from two repositories. The first repository contains three versions of the UML metamodel[10]. The second repository comes from the Graphical Modeling Framework (GMF)[11], an open source project for developing graphical modeling editors. GMF consists of two main metamodels, namely `gmfgraph` that defines the graphical notations and `gmfmapping` that maps domain models, graphical notations, and tool definitions. A description of GMF and its history can be found at [23]. We extracted the metamodel versions from the version control system of GMF as previously performed in [23]. The repository contains 11 versions of the `gmfgraph` metamodel and 16 versions of the `gmfmapping` metamodel. The metamodels for UML and GMF are of similar sizes with 44 classes, 61 associations, and 29 attributes on average.

*Domain-specific comparison of Class Diagram Refactoring.* We applied DSMCompare on the Ecore metamodel to compare different versions of the Ecore models for UML and GMF. The original metamodel *EcoreMM* consists of 20 classes, 48 associations, and 33 attributes. With DSMCompare, we generated the difference metamodel *EcoreDiffMM* with 85 classes, 165 associations, and 174 attributes. The rule metamodel *EcoreRuleMM* contains one more class and association, with 346 attributes.

Table 5 describes the consecutive model versions we considered for each project. We chose the versions that had at least two differences between each consecutive version. The semantic differencing rules we applied are the same rules as for the MM-Refactoring experiment in Section 6.3. However, the four rules related to refactoring methods (rename, pull-up, push-down, and move) are not applicable to the Ecore models. Therefore, we considered 16 semantic differencing rules.

*Results.* Table 5 reports the results of applying DSMCompare in a similar fashion as for the Arduino case study. For example, the fine-grained $Diff_{12}$ model between the Ecore models of `gmfgraph` versions 1.29 and 1.30 reports 59 fine-grained differences. We found nine applicable semantic differencing rules. Among them, the *"extractSuperClass"* rule, which removes attributes from a class and creates a new parent class containing these

[10] `https://git.eclipse.org/c/uml2/org.eclipse.uml2.git/tree/plugins/org.eclipse.uml2.uml/model`
[11] `https://www.eclipse.org/modeling/gmp/`

Table 5: Comparison of model versions for class diagram refactoring from different repositories

| Project | Version n | Version n+1 | Fine Diffs | Semantic Diff | Occurrences | Remaining Fine Diffs |
|---|---|---|---|---|---|---|
| UML | UML 1.4.2 | UML 2.0 | 250 | renameAttribute<br>renameReference<br>mergeReference<br>moveAttribute<br>moveReference<br>removeMiddleMan | 1<br>3<br>2<br>10<br>9<br>7 | 215 |
| GMFgraph | gmfgraph V-1.29 | gmfgraph V-1.30 | 59 | extractSuperClass<br>specializeSuperType<br>pushFeature<br>imitateSuperType<br>generalizeAttribute<br>specializeReferenceType<br>deleteFeature<br>generalizeReference<br>makeContainment | 3<br>6<br>4<br>1<br>1<br>4<br>4<br>1<br>1 | 46 |
| GMFmappings | gmfmappings V-1.45 | gmfmappings V-1.46 | 12 | extractSuperClass<br>makeFeatureVolatile | 1<br>1 | 11 |
| | gmfmappings V-1.48 | gmfmappings V-1.49 | 10 | extractSuperClass<br>pushFeature | 1<br>2 | 7 |
| | gmfmappings V-1.51 | gmfmappings V-1.52 | 2 | replaceEnum | 2 | 2 |
| | gmfmappings V-1.55 | gmfmappings V-1.56 | 2 | replaceInheritanceByDelegation | 1 | 1 |

attributes, occurs in three consecutive model versions, on a total of five matches.

Like for the Arduino case, we see that the fine-grained $Diff_{12}$ model contains fewer changes (cf. last column in Table 5) after applying the semantic differencing rules. For example, in the UML project, $Diff_{12}$ shows that one association is added, while another one is removed, and the type of the association is modified. These fine-grained changes can be encapsulated in the SDRule *"mergeReference"* which corresponds to the intention of the change. As a result of this rule application, fine-grained differences are filtered.

Table 5 shows that DSMCompare is able to extract semantic differences from fine-grained differences. In some cases, it reports more than one match of the same semantic difference, e.g., *"moveAttribute"* (10 times in `UML`) or *"specializeSuperType"* (6 times in `GMFgraph`). In the latter case the rule filters a fine-grained difference at every match, thus presenting less irrelevant information to the user. However, not all rules have filters. For example, *"replaceEnum"* does not filter elements. Nevertheless, by adding semantic difference objects, DSMCompare lifts the user's understanding of changes closer to her intentions: attributes are replaced by enumerations. DSMCompare is also able to find the expected semantic differences (according to the commit messages). For example, it detected the *"extractSuperClass"* rule in both the `GMFgraph` and the `GMFmappings` projects.

As the results of these two case studies show, reporting domain-specific semantic differences reduces verbosity. To better understand this effect, we calculate the verbosity reduction $VR$ as the percentage of diffs eliminated:

$$VR = 1 - \frac{RemainingDiffs}{FineDiffs}$$

where $FineDiffs$ is the number of fine-grained diffs, and $RemainingDiffs$ is the number of remaining diffs after applying DSMCompare.

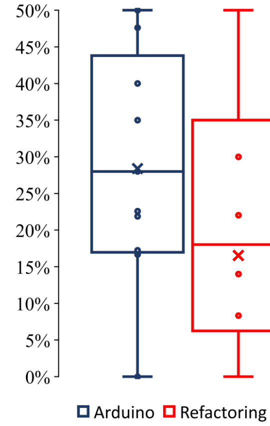


Figure 15: Verbosity reduction for each case study

The box-plot in Figure 15 reports standard descriptive statistics that can be read as follows: the lower bound of the rectangle is the first quartile, the upper bound is the third quartile, the middle bar within the rectangle is the median, the cross is the average, and the top and bottom vertical lines denote the amplitude of the data. Figure 15 reports averages (28% and 16%), medians (28% and 18%), and standard deviations (17%) for both case studies respectively. ArduinoDesigner models contain fewer elements and the differences reported fewer fine-grained differences, which may explain higher verbosity reductions overall. We note that $VR \in [0, 0.5]$ in these projects; thus, using SDRules reduces the fine-grained differences reported by up to a factor of two.

Figure 16 reports the ratio of SDRules per occurrence. For example, 17% of the Arduino and Refactoring SDRules recur four times. We counted a rule as recurring if it matches multiple times on the same model or if it is present in multiple versions. We notice that SDRules occur multiple times and across different projects in each case study. On average, each rule occurs around three times. Also, the majority of the SDRules occur at least

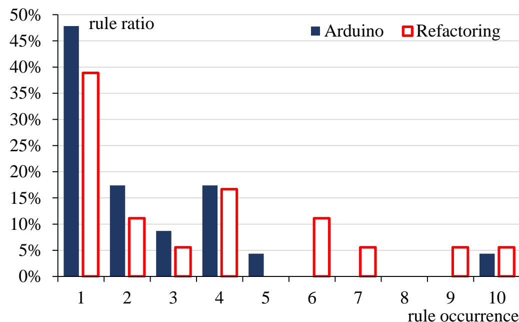


Figure 16: Ratio of the number of rules per number of occurrences for the two case studies

twice for each case study (52% for Arduino and 61% for Refactoring). This justifies that the chosen rules are appropriate for these DSLs.

### *6.5 Discussion*

Next, we discuss the results by answering the four research questions.

*Verbosity (RQ1).* In general, when a SDRule, is applied, at least one semantic difference object is created (and its relations), thus increasing the number of elements in the difference model. If the rule specifies filters, then the number of fine-grained differences decreases. Therefore, the size of the resulting $Diff_{12}$ model varies significantly depending on which rules are applicable. Quantitatively, verbosity is related to the number of fine-grained differences remaining. However, semantic difference objects reduce the verbosity of the difference model qualitatively. They add a higher level of abstraction by providing a precise meaning, which expresses the exact semantic difference for a collection of low-level generic modifications. The domain expert can then better understand the changes that occurred from one version to another, especially when the differences are reported using the same concrete syntax as the DSL. For example, this is particularly peculiar for the Arduino models where the users are young developers with no notion of object-orientation embedded in the abstract syntax. Showing differences using the hardware notations and code sketches can certainly improve their comprehension of the changes and, ultimately, their productivity.

*Semantic differencing rule priorities (RQ2).* Overall, the priorities assigned to DSRules by DSMCompare yields relatively good results. The order of application of the rules has a significant impact on the verbosity, since the more rules are applied, the more semantic difference objects are added and fewer fine-grained differences remain. Furthermore, our results have shown that when two rules are applicable, DSMCompare favors more precise rules. Ultimately, the difference models presented are more meaningful to domain experts.

We have seen that the priority order is not always optimal for all model instances of a DSL. DSMCompare assigns priorities based on static analysis of the metamodel of the DSL. Finding the optimal ordering would require to analyze the given fine-grained $Diff_{12}$ model. We would pre-compute all the matches of each SDRule and add this information as a heuristic to maximize. This must be performed for every difference model of the DSL. In DSMCompare, we opted to provide a solution that is independent from the $Diff_{12}$ model, thus it needs to be computed only once per DSL. One possible use case is to treat the priorities output by DSMCompare as a default suggestion. The pre-computation could then be offered as a suggestion to the user who could decide to manually modify the priorities.

*Ability to extract semantic differences (RQ3).* The premise of this work relies on the ability to report semantic differences in the difference models. The case studies, based on models developed by third-parties, validate that it is possible to find such semantic differences. In DSMCompare, semantic differences are specified by semantic differencing rules. If every rule only occurred once in the case studies, then they would be too specific for each difference model which means they would have to be specified by the end-users almost every time they use DSMCompare. However, our results have shown that the rules we have extracted from the case studies occur multiple times (cf. Figure 16). This strengthens the view that it is possible to extract semantic differences in practice and that the rules can be specified only once per DSL. Nevertheless, with more $Diff_{12}$ models, the set of SDRules may grow. Domain experts can modify or add more rules incrementally thanks to the generated SDRule editor that uses the same environment and notations as the modeling editor used for the DSL.

*Applicability of semantic differencing rules (RQ4).* We found that SDRules occur in different $Diff_{12}$ models (cf. Figure 16). This means that they are not specific to a given $Diff_{12}$ model, but generally applicable to any $Diff_{12}$ model of the DSL. The semantic difference rules must come from a piece of knowledge within the DSL. Therefore, they cannot be invented and need to be semantically meaningful. This knowledge can originate from the operational semantics if it is an executable model, or refactoring patterns, or some DSL-related knowledge-base. From our experience with DSMCompare, SDRules come from the operational semantics of the DSL (as in Pacman) or known refactoring patterns in the DSL: e.g., code-level (like the sketch in Arduino), class-level (like in Ecore models), or model-level (like in feature models [62]). Additionally, in practice, when the DSL designer observes that specific differences are often recurring, this is a good indicator that this may be elevated to a semantic difference rule, to simplify the comprehension of the diffs by the DSL users. Hence, overall, DSMCompare supports

both top-down (i.e., based on known refactorings of the DSL, or its semantics) and bottom-up (i.e., mined from actual changes) approaches to specify SDRules. Finally, our approach is agnostic of the meta-level of the input artifact. As demonstrated in the evaluations, the input can be the metamodel of a DSL (like Pacman and Arduino) or it can be a meta-metamodel (like Ecore).

### *6.6 Threats to validity*

For the controlled experiment, the main limitation is that there were 12 rules in the Pac-Man case and 20 rules in the MM-Refactoring case. Therefore, it was not possible to generate all the 12! and 20! possible permutations. This limitation prevents us to test all possible combinations of rules for the cases. However, to mitigate this threat, we have selected different orderings. One of the selected orders was generated by the Algorithm 4, another one was the reverse of that order, and a collection of five random orders created so that none have the rules placed in the same position and positioned in a way to maximize the diversity. However, testing manually all the possible permutations was not possible.

Another limitation is related to the models used. We created five models, but maybe there are other models in which the application of the rules in the generated order produce a worse result. We did not test all possible models; instead, we created five models in a way that the matches of rules were diversified. In this way, we emphasize the higher-order, middle-order, and lower-order rules by increasing their number of matches, which covers most of the possible opportunities that can happen in any $\mathit{Diff}_{12}$ model. We expect that most $\mathit{Diff}_{12}$ models will fall in one category of the five $\mathit{Diff}_{12}$ models which we have created, i.e., either have a uniform number of matches, maximize some rules, or minimize other rules.

The third limitation was that the way we built the models and we have created the orderings was based on the order generated by Algorithm 4. What we varied in the algorithm was the position of the rules in the priority order. Other algorithms or heuristics may perform better, for instance taking into account NACs, type hierarchies or the number of matches of each rule on the given difference model. Nevertheless, the algorithm constantly shows good performance for the five models with respect to different orderings.

With respect to the case studies, another threat is the way we have computed the verbosity reduction $VR$ of the difference models. The current formula does not take into account the size of models. For example, while the `servo` project has small models, $VR = 40\%$. In contrast, the models in `trigger.all` are larger, yet $VR = 4\%$. There are also other examples of the opposite effect between model size and $VR$. In general, $VR$ highly depends on the number of matches of SDRules. Therefore, a better value of $VR$ should take into account the occurrences of the rules. However, since this number is very small and similar (0-3) in our dataset, this would not influence our results.

Finally, we created the semantic diff rules for Arduino, since (naturally) these were not available from its developers. We compared each two consecutive commits to abstract the multiple atomic changes to a meaningful semantic difference. However, the SDRules we derived may not have been the intention of the original modification. We mitigated this threat by relying on the commit messages, which may indicate that it was the intention of the modeler. Finally, we were able to use DSMCompare on two languages and model histories built by third parties. However, the use of other case studies is required for a stronger validation of our approach.

## 7 Related work

This section reviews related works on model differencing. The survey in [55] presents several model comparison approaches and applications. Model differencing involves *calculation* of the matching model elements, *representation* of their differences, and *visualization* of the differences. Hence, we structure this section paying attention to these three aspects, and also review control version systems for modelling artefacts.

*Model matching calculation.* Kolovos et al. [30] survey current approaches for model matching. These can be: *static identity-based*, which assume a unique identifier for objects; *signature-based*, which compare objects based on a dynamic signature calculated from the objects' properties; *similarity-based*, which match objects based on the weighted similarity of their properties, but obviates the model semantics; and *language-specific*, developed ad-hoc for a modeling language and its semantics. For example, using *signifiers* [35] (i.e., combinations of features of a metamodel class) as a comparison criterion falls in the signature-based category, EMFCompare is similarity-based but permits defining custom matching algorithms, and UMLDiff [67] is language-specific. In general, each solution is a better fit for certain kinds of problems: a language-specific matching algorithm may be faster and more accurate than a generic algorithm, but its implementation requires more effort.

Maoz et al. [38] argue that existing model differencing approaches are purely syntactic and challenge the community to develop semantic diff operators. These calculate a set of diff witnesses that give proof of the real change between two models and the effect on their semantics. Two models may be syntactically different but have no diff witnesses, meaning that they are semantically equivalent. For example, a diff witness of two class diagrams would be an object diagram that is an instance of one of the class diagrams but not of the other, while for activity diagrams, it would be an execution

trace admitted by only one of the diagrams. Diff witnesses also allow deciding whether the semantics of two versions of a model are equivalent, incomparable, or one refines the other. This approach was later realized in the Diffuse framework [37]. Extending our approach to deal with model diffs concerned with the instantiability or executability of models as a comparison criterion is left for future work.

Some researchers have dealt with N-way matching [24, 49], especially in the context of extracting a product line out of a set of structurally similar model variants. In this case, N-way matching is needed to identify the common parts of the involved artefacts. We plan to extend DSMCompare to capture changes between more than two models, and so in this context, it could be used to better understand the (semantic) differences between several model variants.

*Representation of model differences.* Cicchetti et al. [14] propose an approach to represent model differences that is metamodel independent and agnostic of the difference calculation method. Specifically, given two models conforming to the same metamodel, their difference is expressed as another model that conforms to a new metamodel. This new metamodel is derived from the original one by a transformation and allows representing model changes (additions, deletions, and changes). Such difference models induce transformations to translate from one model version to the other and can be composed. While this approach to represent model differences is similar to our proposal, it only works at the abstract syntax level, whereas we also deal with the concrete syntax and support domain-specific patterns to visualize the model differences.

Our approach extends the metamodel of the DSL to represent semantic differencing rules for domain-specific model differences. A related technique is the ramification of metamodels for domain-specific model transformations [31]. In this approach, graph transformation rule patterns are expressed in a domain-specific way. The metamodel of the patterns is generated by transforming the metamodel of the input/output DSLs: relaxing cardinalities, adding transformation-specific attributes and other concepts, and modifying attribute types.

Since low-level differences returned by generic comparison tools may be difficult to understand, Kehrer et al. [26] perform a semantic lifting of such differences to the level of editing operations. For this purpose, low-level differences are represented as models, so that the identification of editing operations consists of finding groups of related low-level changes. This search is performed by rules that are automatically derived from the rule-based specification of the editing operations. Hence, the notion of semantic lifting is similar to our rules for expressing domain-specific semantic differences. However, semantic lifting only deals with the abstract syntax of models, whereas we consider the concrete syntax as well. Similar to semantic lifting approaches such as [20, 65] we identify complex change patterns from low-level changes involved in a metamodel evolution. Although these patterns resemble the rules in our approach, they are generic and predefined. In contrast, our approach allows the DSL engineer to define the semantic differencing rules.

*Visualization of model differences.* Gleicher [21] provides general guidelines for visualizing comparisons. For many different domains, comparing artifacts is a common task and visualizing the comparison often helps. Generally, the visual comparison is displayed using juxtaposition (e.g., as EMFCompare does in Figure 3), superposition, or explicit encoding (like we do in Figure 10).

Brosch et al. [11] visualize the changes and conflicts in concurrently evolved versions of the same UML model using UML profiles (stereotypes and tagged values). This permits modelers to resolve the conflicts within the UML editor of their choice while using the concrete syntax of the manipulated language. However, this approach is only suitable for UML models whereas we pursue a general approach for arbitrary domain-specific languages.

More similar to our work, the authors in [50] focus on the visualization of diagram differences in the diagrams themselves. The rationale is to help users to understand the modifications immediately. Their proposed visualization includes pop-ups reporting the changes performed in the neighborhood, zooming to changes, collapsing irrelevant parts, and using different colors to represent additions (green), deletions (red), and changes (blue), either in a single diagram or confronting two diagram versions. They have developed a tool that uses EMFCompare for model comparison, as we do. However, their tool only permits visualizing atomic changes, represented by different colors. Instead, we support both fine-grained and coarse-grained domain-specific patterns of change. Furthermore, the visualization associated with each pattern is highly configurable. Other works, such as [39, 45], only permit showing changes using different colors or shape styles.

A few works deal with the scalable visualization of differences in the case of large models. To solve this problem, van den Brand et al. [64] combine a generic visualization framework for metamodel-based languages to show the fine-grained differences, with polymetric views that provide support for zooming and filtering. Wenzel [66] also relies on polymetric views to support scalable visualization of differences based on model metrics. Both works are complementary to ours: whereas we provide domain-specificity to the visualization, these other works add a general visualization layer on top.

*Version control systems for models.* Even though models are frequently persisted as text files, the use of traditional text-based version control systems is suboptimal, as we have argued in the introduction. This way, several model repositories with support for version control have

been proposed along the years [4]. The ModelCVS [25] and the AMOR projects [3] proposed dedicated version control systems for models with sophisticated functionalities, like a recommender of possible resolutions for model conflicts [9]. In this setting, DSMCompare could be useful to help understand better the differences between the models, before choosing a resolution strategy.

The model repository of Espinazo-Pagán and García-Molina [19] uses a MySQL database for storage, and a special encoding of model versions to improve efficiency. For a better performance, the authors later proposed the use of NoSQL databases for persistence [18]. EMFStore [29] and CDO [13] are well-known model repositories for EMF, which support collaborative editing and versioning of models. DSMCompare could be used atop these repositories to enable the visualization of (semantic) diffs using the graphical concrete syntax of the DSL.

Commercial modeling tools feature different levels of versioning and model differencing capabilities. LabView has a built-in revision control system that allows to programmatically compare different models [32]. MetaEdit+ [27] features a version control mechanism called Smart Model Versioning [40], which allows comparing models – graphically, textually or by means of a tree – and storing them on any major version control system such as Git. MPS [43] integrates with Git and Subversion and provides some capabilities for viewing model differences, in a textual way [42]. Simulink supports comparing models and highlighting the differences in the original models. Simulink uses a scoring algorithm to determine if two model elements are a match [53]. Similarly, SystemWeaver [61] provides versioning capabilities at the model element level. This way, users can compare an element, view its history, and replace one version of an element with another. While these tools offer different ways to diff models, these are typically fixed and not customizable. Instead, our approach could be valuable here to provide domain-specific, customizable visualizations of the model differences, in a graphical way.

Our approach is based on Eclipse Modeling Framework (EMF). This is a relevant technology, since Eclipse is widely used in MDE research and many companies use Eclipse and EMF tools [1]. Large companies such as IBM are spearheading MDE through EMF [41].

Model differencing and collaborative modeling can lead to clones and duplicates. Some approaches have addressed this problem. Störrle has developed a number of heuristics and algorithms to detect clones in models [56, 57]. Babur et al. [44] leveraged natural language processing, feature extraction and clustering techniques to detect clones in models. We have not focused on detecting model clones in our approach, which is left as future work.

Altogether, to the best of our knowledge, ours is the first comprehensive approach that handles both fine-grained and coarse-grained domain-specific model differences both at the abstract and concrete syntax levels. Moreover, our approach supports the visualization of changes on an automatically modified editor that reuses the graphical concrete syntax of the DSL.

## 8 Conclusion

We have presented a comprehensive approach to represent domain-specific model differences at the abstract and concrete syntax levels. The approach is based on the automated modification of the DSL metamodel to represent fine-grained differences, on the specification of semantic differencing rules to model recurring changes (based on an automatically generated editor), and on the graphical representation of changes using the DSL syntax (by automatically modifying the DSL concrete syntax specification). We have realized our approach in a tool, DSMCompare, that integrates within the Eclipse Modeling Framework and is able to deal with graphical concrete syntaxes specified with Sirius.

Our experience on multiple case studies (Pacman game configuration, Arduino modeling, and metamodel refactoring) and experiments have shown the practicality of our approach to representing meaningful model differences in a domain-specific fashion. With DSMCompare, domain experts can visualize changes using the concrete syntax of the DSL as well as semantically meaningful changes to the domain. This results in less verbose differences that are of tremendous value to the domain experts. We plan to validate this claim with a controlled experiment with users.

We are also considering extending the approach to capture changes between more than two models. To support three-way differencing, we can rely on the three-way merge functionality that EMFCompare offers. We would then extend the DiffMM to support the provenance of each difference. For the SDRules, we need to consider the conflicting situations that may arise when a diff element has at least three different values. The rest of the infrastructure of DSMCompare would only require minimal adaptation when matching and applying SDRules. Although DSMCompare could theoretically support comparing more than three model versions, EMFCompare does not support it. Thus we would need to explore other solutions to address this challenge.

On the tooling side, we will improve the visualization of differences organizing them in layers (e.g., to hide fine-grained differences and visualize only semantic differences). We also plan to incorporate our approach within model repositories, like MDEForge [6], or version control systems for code, like GitHub.

## References

1. D. Akdur, V. Garousi, and O. Demirörs. A survey on modeling and model-driven engineering practices in the

embedded software industry. *Journal of Systems Architecture*, 91:62–82, 2018.

2. A. Al-Herz and A. Pothen. A 2/3-approximation algorithm for vertex-weighted matching. *Discrete Applied Mathematics*, 2019.
3. K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer. AMOR – towards adaptable model versioning. In *Workshop on Model Co-Evolution and Consistency Management*, 2008.
4. K. Altmanninger, M. Seidl, and M. Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.
5. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Model Driven Engineering Languages and Systems*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.
6. F. Basciani, J. Rocco, D. Di Ruscio, A. Salle, L. Iovino, and A. Pierantonio. MDEForge: an extensible web-based modeling platform. In *International Workshop on Model-Driven Engineering on and for the Cloud*, volume 1242, pages 66–75. CEUR-WS.org, 2014.
7. E. Biermann, C. Ermel, and G. Taentzer. Formal foundation of consistent emf model transformations by algebraic graph transformation. *Software & Systems Modeling*, 11(2):227–250, 2012.
8. P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. An introduction to model versioning. In *SFM*, volume 7320 of *LNCS*, pages 336–398. Springer, 2012.
9. P. Brosch, M. Seidl, and G. Kappel. A recommender for conflict resolution support in optimistic model versioning. In *SPLASH/OOPSLA Companion*, pages 43–50. ACM, 2010.
10. P. Brosch, M. Seidl, K. Wieland, and M. Wimmer. We can work it out: Collaborative conflict resolution in model versioning. In *European Conference on Computer-Supported Cooperative Work*, pages 207–214. Springer, 2009.
11. P. Brosch, M. Seidl, M. Wimmer, and G. Kappel. Conflict visualization for evolving UML models. *Journal of Object Technology*, 11(3):2:1–30, 2012.
12. C. Brun and A. Pierantonio. Model differences in the Eclipse Modelling Framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.
13. CDO Model repository. `https://www.eclipse.org/cdo/`, last accessed January 2021.
14. A. Cicchetti, D. D. Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, 2007.
15. J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science*, 376(3):139–163, 2007.
16. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.
17. EMF Compare. `https://www.eclipse.org/emf/compare/`, last accessed January 2021.
18. J. Espinazo-Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A scalable approach for persisting and accessing large models. In *Model Driven Engineering Languages and Systems*, volume 6981 of *LNCS*, pages 77–92. Springer, 2011.
19. J. Espinazo-Pagán and J. García-Molina. A homogeneous repository for collaborative mde. In *International Workshop on Model Comparison in Practice*, pages 56–65. ACM, 2010.
20. J. García, O. Diaz, and M. Azanza. Model transformation co-evolution: A semi-automatic approach. In *Software Language Engineering*, volume 7745 of *LNCS*, pages 144–163. Springer, 2013.
21. M. Gleicher. Considerations for visualizing comparison. *Transactions on Visualization and Computer Graphics*, 24(1):413–423, 2018.
22. GMF. `https://www.eclipse.org/gmf-tooling/`, 2019. (last accessed in June 2019).
23. M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice: The history of GMF. In *Software Language Engineering*, volume 5969 of *LNCS*, pages 3–22. Springer, 2009.
24. S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, and B. Vogel-Heuser. Family model mining for function block diagrams in automation software. In *International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools*, volume 2, pages 36–43. ACM, 2014.
25. G. Kappel, E. Kapsammer, G. Kramler, T. Reiter, W. Retschitzegger, and W. Schwinger. Towards a semantic infrastructure supporting model-based tool integration. In *International Workshop on Global Integrated Model Management*, pages 43–46. ACM, 2006.
26. T. Kehrer, U. Kelter, and G. Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Automated Software Engineering*, pages 163–172. IEEE Computer Society, 2011.
27. S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In *Conference on Advanced Information Systems Engineering*, volume 1080 of *LNCS*, pages 1–21. Springer, 1996.
28. S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation.* Wiley, 2008.
29. M. Koegel and J. Helming. EMFStore: a model repository for EMF models. In *International Conference on Software Engineering*, volume 2, pages 307–308. ACM, 2010.
30. D. Kolovos, D. Di Ruscio, A. Pierantonio, and R. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6. IEEE, 2009.
31. T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer. Explicit transformation modeling. In *MODELS 2009 Workshops*, volume 6002 of *LNCS*, pages 240–255. Springer, 2009.
32. LabView. `https://www.ni.com/en-us/support/documentation/supplemental/21/managing-labview-vi-and-application-revision-history.html`, last accessed May 2021.
33. L. Lambers, H. Ehrig, and F. Orejas. Conflict detection for graph transformation with negative application conditions. In *International Conference on Graph Transformation*, volume 4178 of *LNCS*, pages 61–76. Springer, 2006.

34. L. Lambers, D. Strüber, G. Taentzer, K. Born, and J. Huebert. Multi-granular conflict and dependency analysis in software engineering based on graph transformation. In *International Conference on Software Engineering*, pages 716–727. ACM, 2018.
35. P. Langer, M. Wimmer, J. Gray, G. Kappel, and A. Vallecillo. Language-specific model versioning based on signifiers. *Journal of Object Technology*, 11(3):4: 1–34, 2012.
36. Y. Lin, J. Gray, and F. Jouault. DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361, 2007.
37. S. Maoz and J. O. Ringert. A framework for relating syntactic and semantic model differences. *Software & System Modeling*, 17(3):753–777, 2018.
38. S. Maoz, J. O. Ringert, and B. Rumpe. A manifesto for semantic model differencing. In *MODELS 2010 Workshops*, volume 6627 of *LNCS*, pages 194–203. Springer, 2011.
39. A. Mehra, J. C. Grundy, and J. G. Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *Automated Software Engineering*, pages 204–213. ACM, 2005.
40. MetaEdit+. `https://www.metacase.com/news/smart_model_versioning.html`, last accessed May 2021.
41. P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernandez, B. Nordmoen, and M. Fritzsche. Where does model-driven engineering help? experiences from three industrial cases. *Software & Systems Modeling*, 12(3):619–639, 2013.
42. MPS. Differences viewer for files. `https://www.jetbrains.com/help/mps/differences-viewer.html`, last accessed May 2021.
43. MPS. Version control. `https://www.jetbrains.com/help/mps/version-control-integration.html`, last accessed May 2021.
44. Önder Babur, L. Cleophas, and M. van den Brand. Metamodel clone detection with SAMOS. *Journal of Computer Languages*, 51:57–74, 2019.
45. D. Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. In *Proceedings of the 9th European software engineering conference held jointly with 11th international symposium on Foundations of software engineering*, pages 227–236. ACM, 2003.
46. OMG. The Object Constraint Language (OCL) v. 2.4. Specification. `http://www.omg.org/spec/OCL/`, 2014. (last accessed in January 2021).
47. OMG. XMI metadata interchange v. 2.5.1. `https://www.omg.org/spec/XMI/About-XMI/`, last accessed January 2021.
48. R. F. Paige, N. D. Matragkas, and L. M. Rose. Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272–280, 2016.
49. D. Reuling, M. Lochau, and U. Kelter. From imprecise n-way model matching to precise n-way model merging. *Journal of Object Technology*, 18(2):8:1–20, 2019.
50. A. Schipper, H. Fuhrmann, and R. von Hanxleden. Visual comparison of graphical models. In *International Conference on Engineering of Complex Computer Systems*, pages 335–340. IEEE, 2009.
51. D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.
52. F. Schwägerl, S. Uhrig, and B. Westfechtel. A graph-based algorithm for three-way merging of ordered collections in EMF models. *Science of Computer Programming*, 113:51–81, 2015.
53. Simulink. `https://www.mathworks.com/help/simulink/ug/about-simulink-model-comparison.html`, last accessed May 2021.
54. Sirius. `https://www.eclipse.org/sirius/`, last accessed January 2021.
55. M. Stephan and J. R. Cordy. A survey of model comparison approaches and applications. In *Model-Driven Engineering and Software Development*, pages 265–277. SciTePress, 2013.
56. H. Störrle. Towards clone detection in uml domain models. In *European Conference on Software Architecture: Companion Volume*, pages 285–293. ACM, 2010.
57. H. Störrle. Cost-effective evolution of research prototypes into end-user tools. *Science of Computer Programming*, 134:47–60, 2017.
58. D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy. Henshin: A usability-focused framework for emf model transformation development. In *International Conference on Graph Transformation*, volume 10373 of *LNCS*, pages 196–208. Springer, 2017.
59. E. Syriani and H. Vangheluwe. A modular timed graph transformation language for simulation-based design. *Software & System Modeling*, 12(2):387–414, 2013.
60. E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. AToMPM: A web-based modeling environment. In *MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition*, volume 1115, pages 21–25. CEUR-WS.org, 2013.
61. SystemWeaver. `https://support.systemweaver.se/support/solutions/articles/31000156469-versioning-in-systemweaver`, last accessed May 2021.
62. M. Tanhaei, J. Habibi, and S.-H. Mirian-Hosseinabadi. Automating feature model refactoring: A model transformation approach. *Information and Software Technology*, 80:138–157, 2016.
63. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.
64. M. van den Brand, Z. Protić, and T. Verhoeff. Generic tool for visualization of model differences. In *International Workshop on Model Comparison in Practice*, pages 66–75. ACM, 2010.
65. S. D. Vermolen, G. Wachsmuth, and E. Visser. Reconstructing complex metamodel evolution. In *Software Language Enginerring*, volume 6940 of *LNCS*, pages 201–221. Springer, 2012.
66. S. Wenzel. Scalable visualization of model differences. In *Workshop on Comparison and versioning of software models*, pages 41–46. ACM, 2008.
67. Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Automated software engineering*, pages 54–65, 2005.
68. M. Zadahmad, E. Syriani, O. Alam, E. Guerra, and J. de Lara. Domain-specific model differencing in visual concrete syntax. In *Software Language Engineering*, pages 100–112. ACM, 2019.