Domain-Specific Model Differencing in Visual Concrete Syntax

Manouchehr Zadahmad
Université de Montréal
Canada
zadahmad@gmail.com

Eugene Syriani
Université de Montréal
Canada
syriani@iro.umontreal.ca

Omar Alam
Trent University
Canada
omaralam@trentu.ca

Esther Guerra
Universidad Autónoma de Madrid
Spain
Esther.Guerra@uam.es

Juan de Lara
Universidad Autónoma de Madrid
Spain
Juan.deLara@uam.es

Abstract

Like any other software artifact, models evolve and need to be versioned. In the last few years, dedicated support for model versioning has been proposed to improve the default text-based versioning that version control systems offer. However, there is still the need to comprehend model differences in terms of the semantics of the modeling language. For this purpose, we propose a comprehensive approach that considers both abstract and concrete syntax, to express model differences in terms of the domain-specific language (DSL) used and define domain-specific semantics for specific difference patterns. The approach is based on the automatic extension of the DSL to enable the representation of changes, on the definition of rules to capture recurrent domain-specific difference patterns, and on the automatic adaptation of the graphical concrete syntax to visualize the differences. We present a prototype tool support and discuss its application on versioned models created by third parties.

CCS Concepts • **Software and its engineering → Domain specific languages**; *Visual languages*; Software configuration management and version control systems.

Keywords Model-driven Engineering, Model differencing, Graphical concrete syntax

ACM Reference Format:
Manouchehr Zadahmad, Eugene Syriani, Omar Alam, Esther Guerra, and Juan de Lara. 2019. Domain-Specific Model Differencing in Visual Concrete Syntax. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19), October 20–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357766.3359537

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, like the UML, but the use of domain-specific languages (DSLs) is also frequent [18, 31].

Like other software artifacts involved in a development process, models evolve [29] and, therefore, need to be versioned to have a record of their changes [3]. Sometimes, models are persisted as text files (e.g., using XMI), which permits using code version control systems on them. However, text-differencing is not adequate for models as it may report irrelevant model differences (e.g., objects that appear in different file positions). For this reason, the modeling community has proposed specific model versioning systems [1, 7, 12, 26] and approaches for model differencing [9], conflict resolution, and merging [4, 32].

A crucial aspect of versioning systems is the possibility to visualize matches and differences of the history of a model in a comprehensible manner. However, many approaches, like EMFCompare [9], represent the differences between two versions of a model using low-level generic traces that may be difficult to comprehend. Moreover, these traces typically are at the level of the abstract syntax, which may hinder their understanding. Therefore, we propose to represent traces in a domain-specific way, assign a domain-specific semantics to recurring model differences, and visualize those differences at the concrete syntax level. To ensure the practicality of our proposal, we provide automated tool support to minimize the effort of applying the approach to arbitrary DSLs. In this paper, we focus on graphical concrete syntaxes realized through the Sirius framework [33].

The contributions of this paper are the following. First, we propose a method to encode model differences as a single domain-specific model. This is achieved by automatically extending the DSL meta-model with domain-specific change operations. Second, we propose a means to create higher-level representations of lower-level differences using rules. Third, we provide an automated way to represent model differences using the DSL graphical concrete syntax. Finally, we provide a prototype tool support, able to adapt automatically Sirius-based editors for model change visualization, and use it to validate our proposal based on graphical DSLs and model histories created by third-parties.

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 in terms of the DSL. This encompasses the semi-automated extension of the DSL meta-model and its concrete syntax. For the latter, we use Sirius as an illustration. In Section 4, we detail how to define domain-specific higher-level change descriptions. In Section 5, we evaluate the approach by its application to the visualization of differences of model histories in an open source project. Finally, we compare with related work in Section 6 and conclude the paper in Section 7.

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 contains the compared pair of elements. The diff set illustrates how much a model element in the match is different from its pair, i.e., each match can have multiple diffs. The most popular generic model comparison tools, EMFCompare [9] for instance, produces 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 granular changes in the model. Consequently, the comparison tool shows the user all low-level changes, such as an association between two objects deleted, 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 concepts 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

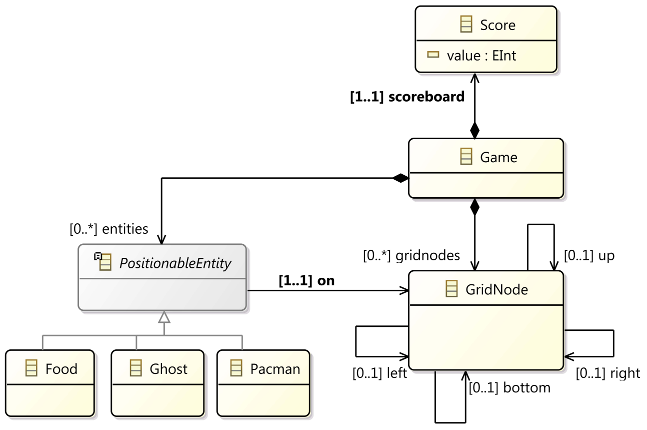


Figure 1. The metamodel of the Pacman DSL (MM)

[16] or by using a metamodel to represent model differences [8]. 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 of a DSL). Furthermore, the existing approaches do not take domain-specific model semantics into consideration during the comparison process.

To address these issues, we introduce an approach, called *DSMCompare*, that 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 [35]. 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 granular 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 domain-specific 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

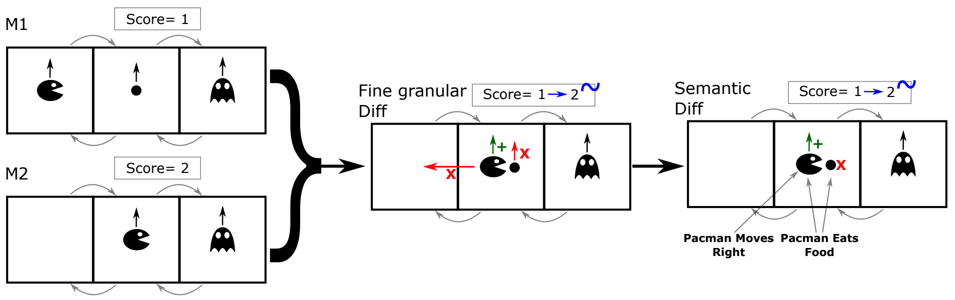
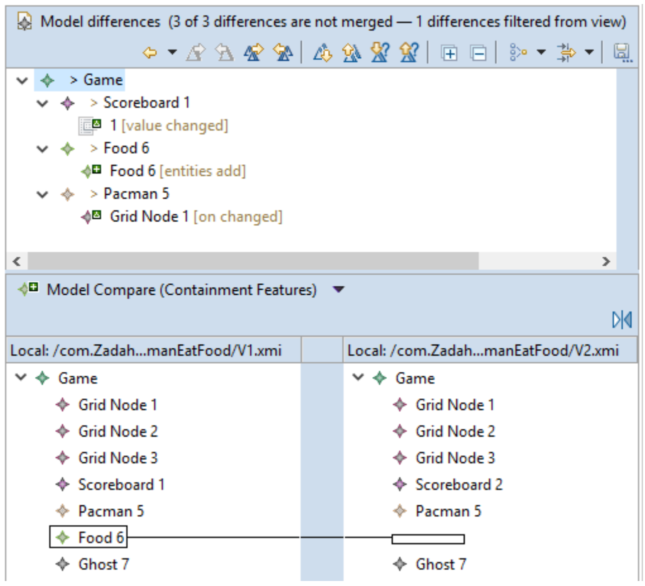


Figure 2. Running example using DSMCompare



Model differences (3 of 3 differences are not merged — 1 differences filtered from view)

```
> Game
  > Scoreboard 1
      1 [value changed]
  > Food 6
      Food 6 [entities add]
  > Pacman 5
      Grid Node 1 [on changed]
```

Model Compare (Containment Features)

Local: /com.Zadah...manEatFood/V1.xmi	Local: /com.Zadah...manEatFood/V2.xmi
Game	Game
Grid Node 1	Grid Node 1
Grid Node 2	Grid Node 2
Grid Node 3	Grid Node 3
Scoreboard 1	Scoreboard 2
Pacman 5	Pacman 5
Food 6	
Ghost 7	Ghost 7

Figure 3. Representation of difference model in EMFCompare for the Pacman game DSL

which gets deleted, and the scoreboard value is incremented. The final difference model $Diff_{12}$ is depicted at the right of Figure 2.

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. This is useful for two types of users: DSL engineers and DSL users. The DSL engineer creates a metamodel *MM* (abstract syntax) and the

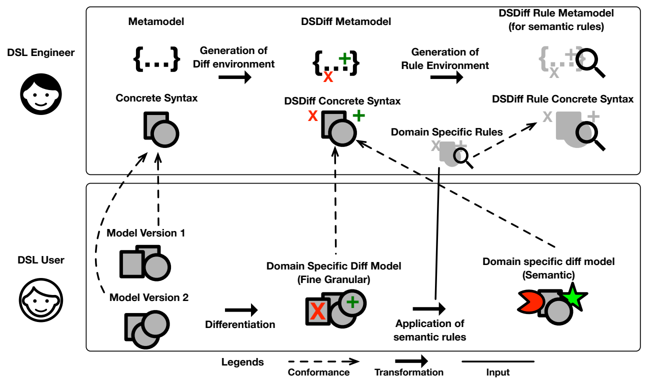

DSL Engineer
Metamodel
Generation of Diff environment
DSDiff Metamodel
Generation of Rule Environment
DSDiff Rule Metamodel (for semantic rules)
Concrete Syntax
DSDiff Concrete Syntax
DSDiff Rule Concrete Syntax
Domain Specific Rules
DSL User
Model Version 1
Model Version 2
Differentiation
Domain Specific Diff Model (Fine Granular)
Application of semantic rules
Domain specific diff model (Semantic)
Legends
Conformance
Transformation
Input

Figure 4. Overview of the approach

concrete syntax *CS* to define the DSL. 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 approach produces a domain-specific diff metamodel *DSDiffMM* and concrete syntax *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. The approach also produces an environment to describe high-level semantic differences in the form of rules tailored to the DSL. Namely, it produces a domain-specific rule metamodel *DSRuleMM* and concrete syntax *DSRuleCS*, to allow the DSL engineer to define the set of rules to apply on $Diff_{12}$. As discussed previously, having domain-specific rules is important to reason 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 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 model version. Instead of showing that an operation is simply being added in the second version, DSMCompare

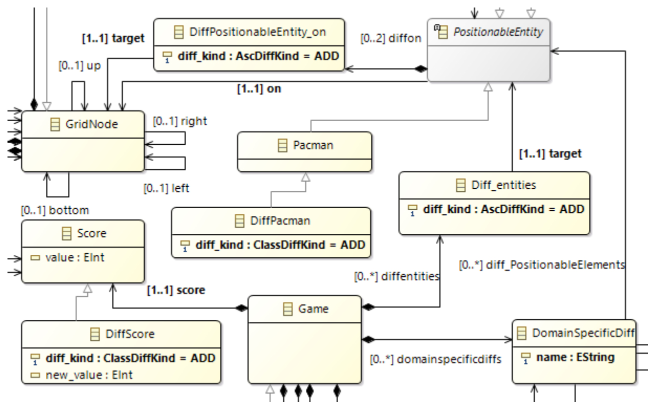


Figure 5. Excerpt of the generated difference metamodel

uses the rule to represent this change as the second operation overriding the first one.

The DSL user can use our approach 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 EMFCompare. 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 granular differences in the concrete syntax of the DSL. With a library of rules predefined by the DSL engineer, the approach applies the applicable rules on $Diff_{12}$ to produce a semantically lifted difference model.

3 Fine Granular 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.

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 for domain-specific difference *DSDiffMM* (see Figure 5) based on *MM* (see Figure 1). Algorithm 1 outlines the transformation from the latter to the former. 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 by the kind of difference. To represent a difference in an object of a class, like `Score`, we create a subclass with an additional attribute `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.

Algorithm 1 Transformation from MM to DSDiffMM

```
procedure GenerateDSDiffMM(MM)
  DSDiffMM ← MM.clone("DSDiffMM")
  DSDiffMM.createEnum("ClassDiffKind", {ADD,DEL,MOD})
  DSDiffMM.createEnum("AscDiffKind", {ADD,DEL})
  for all class C in DSDiffMM do
    if not C.isAbstract() then
      DiffC ← DSDiffMM.createClass("DiffC")
      DiffC.setSuperClass(C)
      DiffC.addAttribute("diff_kind", ClassDiffKind)
    end if
    for all attribute a in C.getAllUniqueAttributes() do
      DiffC.addAttribute("new_a", a.getType())
    end for
  end for
  for all association S in DSDiffMM do
    C1 ← S.getSource(), C2 ← S.getTarget()
    if C1 ≠ DSDiffMM.getRootClass() then
      DiffC1_S ← DSDiffMM.createClass("DiffC1_S")
      DiffC1_S.addAttribute("diff_kind", AscDiffKind)
      n ← S.getTargetCardinalities().target().upperBound()
      if S.isComposition() then
        diffS ← C1.addComposition("diffS", DiffC1_S)
      else
        diffS ← C1.addAssociation("diffS", DiffC1_S)
      end if
      diffS.setCardinalities(1..1, 0..2×n)
      target ← DiffC1_S.addAssociation("target", C2)
      target.setCardinalities(0..1, 1..1)
    end if
  end for
  DSDiff ← DSDiffMM.createClass("DomainSpecificDiff")
  DSDiff.addAttribute("name", String)
  for all class C in DSDiffMM do
    diff_C ← DSDiff.addAssociation("diff_C", C)
    diff_C.setCardinalities(1..1, 0..*)
  end for
  R ← DSDiffMM.getRootClass()
  diffs ← R.addComposition("diffs", DSDiff)
  diffs.setCardinalities(1..1, 0..*)
  return DSDiffMM
end procedure
```

Note that this procedure does not transform class inheritance. 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. 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 granular differences in $Diff_{12}$. To enable the representation of semantic differences, the procedure creates a DomainSpecificDiff class that holds the name of the semantic difference that a combination of original and *DSDiff* classes represent. This will be used in the second step when applying domain-specific rules.

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

When the DSL user 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 *DSDiffCS*. Since the DSL engineer has defined a concrete syntax *CS* for the DSL, he should also provide one for *DSDiffMM*. 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. The DSL engineer can then customize it if so desired. In this subsection, we describe how to generate *DSDiffCS* from *CS*, assuming a graphical concrete syntax.

Sirius [40] 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 [15], MetaEdit+ [17] and AToMPM [36], our description is based on Sirius as 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) [27], can filter visualizations depending on a condition. Finally, it is possible to define a palette of icons to instantiate *MM* classes and associations by customizing the ToolSection in Sirius.

We generate *DSDiffCS* by means of an outplace transformation 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

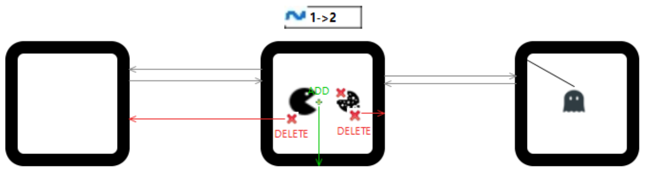


Figure 6. Fine granular difference model Diff_{12} of *M1* and *M2*

deleting, annotated similarly to Nodes. The `target` association is rendered by an `EdgeNode`.

The only element in *DSDiffMM* that does not have a visualization in *CS* is the `DomainSpecificDiff` class. By default, we represent it with a rectangle with its `name` attribute value displayed inside. Note that if *CS* makes use of icon files to render the elements of *MM*, the DSL engineer must also provide a set of icon files for each `Diff_` class and association. The transformation assumes the naming convention corresponding to the name of the class to map an icon to each node of *DSDiffCS*. 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 Diff_{12} 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, [22] dynamically computes an identifier for each model element based on their properties (e.g., type and attribute values). Alternatively, metamodel-agnostic approaches, like [6, 8], 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 will assume that the result from a difference algorithm outputs a list of differences for classes Δ_C and for associations Δ_A separately, such as in EMFCompare. We denote an element $E' \in \Delta_C$ using primed uppercase letters. This way, if E' is a deletion

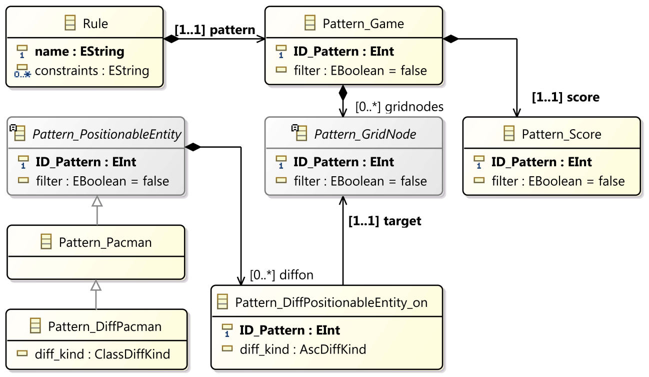


Figure 7. Excerpt of the domain-specific difference rule metamodel *PacmanRuleMM*

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 in Figure 6.

4 Domain-specific Differencing

In this section, first we outline how to generate a graphical environment for the DSL engineer that allows creating domain-specific difference rules. We then show how to apply the rules to aggregate fine granular differences in a meaningful manner that follows the intention of the domain.

4.1 Rules for domain-specific differences

As explained in Section 2, we automatically derive an environment for specifying domain-specific difference 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 granular differences and replace it with a `DomainSpecificDiff` class that was created in Algorithm 1.

Our domain-specific difference rules act similarly to inplace model transformation rules [11] with a precondition and a postcondition component. Algorithm 2 outlines the procedure to produce *DSRuleMM* from *DSDiffMM* and Figure 7 shows the result. It is inspired from [20] where they produce domain-specific model transformation rule patterns from a DSL.

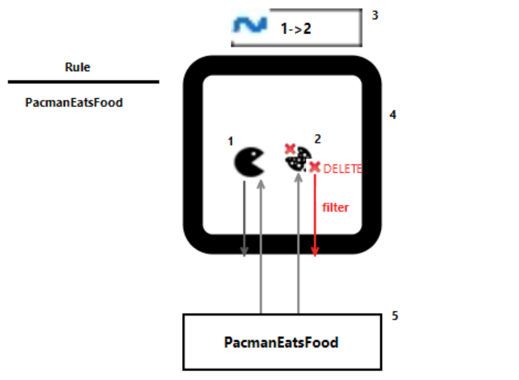
Algorithm 2 Transformation from `DSDiffMM` to `DSRuleMM`

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

As in Algorithm 1, this procedure starts by reusing all the elements of *DSDiffMM* and adapts them to the new needs. Every class and association are prefixed with `Pattern_`. 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 for all pattern classes. A unique identifier distinguishes instances of the same classes to facilitate writing constraints. A *filter* attribute is used to signify that the element in $Diff_{12}$ should be removed when applying the rule. This filters out fine granular differences when a domain-specific difference is more meaningful.

Finally, we add a new `Rule` class as the new root class of the metamodel. This enables the transformation engine to navigate easily through the elements of the rule. The `Rule` class allows specifying a list of constraints over attribute values. In practice, constraints are written in Java and executed dynamically using BeanShell[1], an embedded interpreter to run Java scripts. Within constraints, pattern objects can be accessed using their identifier. For instance, the constraint `Item(3,'value') < Item(3,'new_value')` states that the new value of the score should be greater than the original value in rule `PacmanEatsFood` in Figure 8.

[1]https://github.com/beanshell/beanshell



```
Constraints = ["Item(3,"value") < Item(3,"new_value")"]
```

Figure 8. The domain-specific difference rule for `PacmanEatsFood`

4.2 Automatic generation of a visual environment for domain-specific 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 domain-specific rules in the same language workbench.

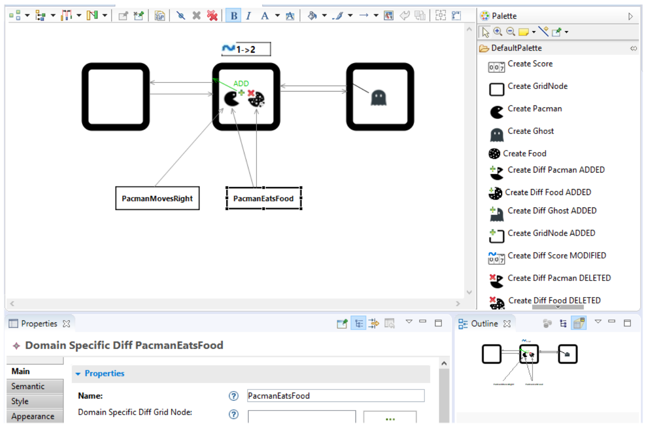
For this purpose, we transform the *DSDiffCS* model into the concrete syntax for rules *DSRuleCS*. The transformation is very similar to the one described in Section 3.2. We copy the viewpoint specification model and adapt it to the *DSRuleMM*. Each `NodeMapping` displays *"filter"* if the `filter` attribute is set to true, as well as the `pattern_id` of the object. All other attribute values from their *DSDiffMM* counterpart are removed as they are no longer present in pattern classes, like in the `Score`.

Figure 8 illustrates a rule in the generated domain-specific environment. The rule describes that a `PacmanEatsFood` change occurs when `Pacman` is on a grid node, a food is deleted from that same node, and the score is incremented.

4.3 From fine granular to domain-specific differences

As outlined in Figure 4, we apply the domain-specific rules to enhance the fine granular difference model $Diff_{12}$ with semantic differences. Given the difference model $Diff_{12}$ produced as described in Section 3.3, we implemented an inplace transformation that applies the rules on $Diff_{12}$.

Inspired by inplace graph transformations [11], the algorithm has a matching phase to find an occurrence of the precondition of the rule and a rewriting phase to apply the postcondition of the rule. In a domain-specific rule, the precondition consists of the constraints of the rule and the structure formed by the pattern objects contained inside the rule except for the `DomainSpecificDiff` object. The postcondition of the rule is specified by the `DomainSpecificDiff` instance and its `diff_` associations (see lines 31–36 of Algorithm 1), along with all `filter` attributes that are set to `true`



Domain Specific Diff PacmanEatsFood

Properties

Name: PacmanEatsFood

Domain Specific Diff Grid Node:

Figure 9. The domain-specific difference for PacmanEats-Food in the generated editor after applying the rules

in the pattern classes. For example, the PacmanEatsFood 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 DomainSpecificDiff object named PacmanEatsFood and hides the deleted DiffPositionableElement_on link associated with DiffFood.

The matching phase starts from the root element (e.g., Game or DiffGame) and traverses the *Diff*$_{12}$ model following the pattern elements of the rule. Our implementation follows a depth-first search strategy with backtracking. To slightly optimize the traversal, *DSDiffMM* classes have priority over *MM* classes. When a complete match of the pattern is found, we evaluate the list of constraints. Once a pattern that satisfies the constraints is found, we apply the rewriting phase according to the postcondition of the rule. The DomainSpecificDiff object is created in *Diff*$_{12}$ with its diff_ associations. All objects marked as filtered in the pattern are removed. Altogether, the resulting *Diff*$_{12}$ model is *semantically lifted* to show higher-level differences that are deemed important and meaningful to the DSL user. Applying the rule 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 9 illustrates the final difference model provided by our approach. It shows that Pacman has moved right and eaten the food: an example where two rules have been applied. Altogether, compared to Figure 3, the DSL user can inspect the domain-specific changes in an editor that resembles the one he used to manipulate the original models *M1* and *M2*.

As a current limitation, our rules do not support matching a subclass of a pattern class: in *DSDiffMM*, the DiffScore class inherits from the Score class. Furthermore, abstract classes from *MM*, like PositionableElement, cannot be used when specifying patterns. As this can be useful to define more general rules [10], we plan to provide support for this feature in future work.

A rule may find more than one match in *Diff*$_{12}$. However, care should be taken since applying a rule may remove elements. In our implementation, we apply the rules iteratively as long as matches are found. However, a rule is applied only once to each match thanks to a temporary dirty flag. Most of the time, there are more than one domain-specific rule specified for a DSL. Our current implementation applies the rules sequentially. As a limitation, if a rule filters an element that is required in the precondition of another rule, the latter will not find a match.

5 Evaluation and Discussion

In this section, we validate the approach with a non-trivial visual DSL developed by a third party (the Arduino Designer) and on existing versioned models, from the Arduino Designer repository on GitHub[2]. First, we describe the case study, and then we present an evaluation aimed to answer these three research questions (RQs):

RQ1 Can we extract semantic diffs from fine granular diffs?
RQ2 Are fine granular diffs more verbose than semantic diffs?
RQ3 Are semantic diffs recurring?

5.1 Case study: Arduino Designer

Arduino Designer is an environment specially tailored to young children, to create simple programs for Arduino[3], 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 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 the GitHub repository, 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

[2]https://github.com/mbats/arduino/
[3]https://www.arduino.cc/

semantic diff rules. The changes in the configuration part typically consist of adding or replacing devices in appropriate pins of the board.

5.2 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 generated 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 similar scale.

The Arduino GitHub repository includes 13 working example projects. We applied DSMCompare on all projects; Table 1 summarizes the results. Each model has between 2 to 6 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 one* and *Version two* columns).

The fourth column (*Fine Diffs*) shows the total number of *class/association* diff pairs found by DSMCompare. For example, as the fifth row of the table shows, in the `fadelight` project, when comparing the version *While* and the version *Sub instructions*, DSMCompare reported 20 fine-granular class differences and 22 association differences. The column *Semantic Diff* shows the title of the semantic diff recognized among the fine-granular diff, and column *Ocurrences* represents the number of occurrences of that semantic diff. Finally, the last column shows the number of fine granular pairs of class/association differences encapsulated in the corresponding semantic diff.

As Figure 10 illustrates, DSMCompare reported two semantic differences of *"Refactor a while loop"* type representing a while-loop refactoring in the `fadelight` project. 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 granular $Diff_{12}$ model represents fewer changes since this corresponds to the changes output by EMFCompare and converted into domain-specific diffs. For example, the `tigger.tail` model adds a music player module to a digital pin of the board and adds instructions to turn it on and off at the beginning and at the end of the main loop. In this case, the fine granular $Diff_{12}$ model shows one class and association removal and two of each added. This change can be encapsulated in a domain-specific difference rule *"Add music player"* which corresponds to the intention of the change.

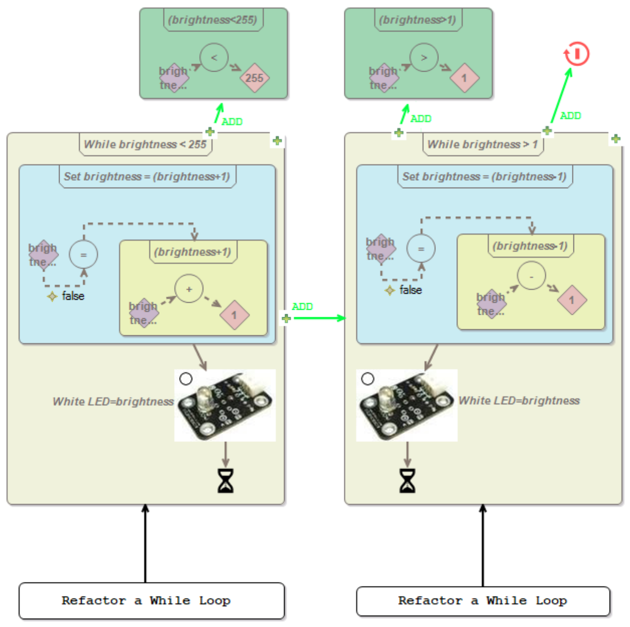


Figure 10. The domain-specific difference in Arduino designer for `fadelight` project

The table also clearly shows that DSMCompare is able to extract semantic diffs from fine granular diffs. Moreover, in some cases, no DSRule matches in a comparison (e.g., `morse-code`), whereas some differencing also result in reporting more than one semantic diff (e.g., `functions`). DSMCompare is also able to find the expected semantic diffs. For example, it detected the *"Refactor a while loop"* semantic diff not only in the `fadelight` project twice, but it also recognized the *"Servo Feature enabled"* semantic diff in both the `infraredsensor` and the `servo` projects. Most of the identified semantic differences are additions to an already designed Arduino model related to fixing errors. 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. Only the `singleinput` project involved a correction, where the devices plugged in the input and output pins were inverted.

Overall, using a case study of a DSL built by a third party, we can aswer **RQ1** by stating that our approach can be used to extract semantic diffs from fine granular ones. Regarding **RQ2**, these semantic diffs use concepts of the domain and are typically much more succinct than the final granular ones, as in 10 out of the 17 cases, semantic diffs are one order of magnitude smaller (while providing domain-specific visualization capabilities). Finally, regarding **RQ3**, as column *semantic diff* shows, we were able to build a library of diff rules to capture recurring semantic changes, like the refactoring of loops, or refactorings to a conditional.

Table 1. Comparison of model versions in the Arduino Designer examples repository

Project	Version one	Version two	Fine Diffs	Semantic Diff	Occurrences	Semantic Fine Diffs
			Class/Association			Class/Association
`/alarmlight`	Initialize	Migrate alarmlight example to sirius	41/16	Refactor a Repeat Loop	3	1/1
				Increase Repeat iteration	2	1/0
	Repeat	Fix Generation For AlarmLight Example	52/8	Refactor a Repeat Loop	1	1/1
				Delete a Repeat Loop	1	1/1
`/fadelight`	While	Sub instructions	20/22	Refactor a While Loop	2	1/2
	Fadelight example	Migrate alarmlight example to sirius	5/3	Incomplete While Loop Deleted	1	1/1
`/functions`	Initialize	Add functions example	39/24	Add a function	1	1/0
				Function Call	2	1/1
`/infraredsensor`	Initialize	Add servo	15/2	Servo Feature enabled	1	2/3
				Support infrared connections	1	1/2
`/morsecode`	Morse code	Migrate morsecode example to sirius	2/0	–	-	-/-
`/reset`	Initialize	Add reset example	6/0	Reset functionality enabled	1	1/2
`/rotationsensor`	Initialize	While	6/0	–	-	-/-
`/servo`	Initialize	Migrate morsecode example to sirius	14/2	Servo Feature enabled	1	2/3
				Support infrared connections	1	1/2
`/simpleinput`	simpleinput	Simple input example	3/0	–	-	-/-
`/tigger.all`	Add Tigger example	Update the tigger example	99/36	Refactor a Repeat Loop	1	1/1
				Refactor to an if condition	1	1/2
`/tigger.bubble`				Refactor to an if condition	1	1/2
	Initialize	Add tigger bubble example	41/14	Refactor a Repeat Loop	1	1/1
				Refactor a Bubble machine	1	9/9
	Add tigger bubble example	Fix issue on bubble example	2/0	–	-	-/-
`/tigger.necklace`				Refactor to an if condition	1	1/2
	Initialize	Add tigger necklace example	36/15	Refactor a Repeat Loop	1	1/1
				A cat necklace added	1	9/9
`/tigger.tail`	Initialize	Update cat tail example to Add miaou sound	13/1	Refactor to an if condition	1	1/2
				Add music player	1	3/2
	Add Tigger tail example	Update tail example	2/2	–	-	-/-

5.3 Discussion and threats to validity

Both Pacman and Arduino projects are examples where providing the differences between two model versions is of tremendous value to the DSL user. Without DSMCompare, differencing reports many changes at the abstract syntax level. 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. We plan to validate this claim with a controlled experiment observing DSL users using DSMCompare or not.

Apart from providing the differences in concrete syntax, the efficiency of our approach depends on the domain-specific rules provided. Typically, these rules come from the operational semantics of the DSL (as in Pacman) or from known refactoring patterns in the DSL: e.g., code-level, like the sketch in Arduino, class-level, like in object-oriented models (EcoreTools[4]), or model-level, like in feature models [37].

Regarding threats to validity, we were able to use DSMCompare on a DSL built by a third party. However, the use of other case studies is required for a stronger validation of our approach. Similarly, the models we analyzed are of moderate size. The models in each project are of similar size: around 38 class instance (objects) and instances of associations (references). GitHub reports the textual differences of the serialized model in XMI. We plan to use our approach with larger models in future work to assess the scalability of DSMCompare.

[4]https://www.eclipse.org/ecoretools/

6 Related Work

This section reviews related works on model differencing (see also [34] for a survey on 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.

Model matching calculation. Kolovos et al. [19] 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 aggregated 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* [21] (i.e., combinations of features of a metamodel class) as comparison criteria falls in the signature-based category, EMFCompare is similarity-based but permits defining custom matching algorithms, and UMLDiff is language-specific. In general, each solution is 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. [24] 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 a 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 [23]. Extending our approach to deal with model diffs concerned with the instantiability or executability of models as comparison criteria is left for future work.

Representation of model differences. Cicchetti et al. [8] 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 rules for domain-specific model differences. A related technique is the ramification of metamodels for domain-specific model transformations [20]. 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 incomprehensible, Kehrer et al. [16] 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 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 [13, 39] 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 domain specific rules.

Visualization of model differences. Gleicher [14] provides general guidelines for visualizing comparisons. For many different domains, comparing artifacts is a common task and visualizing the comparison often helps. Generally, visual comparison is displayed using juxtaposition (e.g., as EMFCompare does in Figure 3), superposition, or explicit encoding (like we do in Figure 9).

Brosch et al. [5] 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 [30] focus on the visualization of diagram differences on the diagrams themselves. The rationale is helping 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 [25, 28], only permit showing changes using different colors or shape styles.

A few works deal with the scalable visualization of differences in case of large models. To solve this problem, van den Brand et al. [38] 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 [41] 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.

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.

7 Conclusion

In this paper, we have presented a comprehensive approach to represent domain-specific model diffs at the abstract and concrete syntax levels. The approach is based on the automated modification of the DSL metamodel to represent fine-granular diffs, on the specification of 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 within the Eclipse Modeling Framework with Sirius. We have shown the practicality of our approach to represent changes of Arduino Designer models.

In the future, we plan to improve the expressiveness of our pattern rules, e.g., adding negative application conditions, and support for abstract objects. We are also considering extending the approach to capture changes between more than two models. We also plan to incorporate our approach within model repositories, like MDEForge [2], or version control systems for code, like GitHub. Finally, we will also conduct a user study to check the usability and utility of the approach.

Acknowledgment

This work is funded by the NSERC Discovery Grant (RGPIN/03749-2015), the Spanish Ministry of Science (RTI2018-095255-B-I00), and the Madrid Region (S2018/TCS-4314).

References

[1] AMOR. last accessed February 2019. Adaptable model versioning. http://www.modelversioning.org/
[2] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio. 2014. MDEForge: an extensible web-based modeling platform. CEUR-WS, 66–75.
[3] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. 2012. An introduction to model versioning. In *SFM (LNCS)*, Vol. 7320. Springer, 336–398.
[4] P. Brosch, M. Seidl, K. Wieland, and M. Wimmer. 2009. We can work it out: Collaborative conflict resolution in model versioning. In *European Conference on Computer-Supported Cooperative Work*. Springer, 207–214.
[5] P. Brosch, M. Seidl, M. Wimmer, and G. Kappel. 2012. Conflict visualization for evolving UML models. *Journal of Object Technology* 11, 3 (2012), 2:1–30.
[6] C. Brun and A. Pierantonio. 2008. Model differences in the Eclipse Modelling Framework. *UPGRADE, The European Journal for the Informatics Professional* 9, 2 (2008), 29–34.
[7] CDO. last accessed February 2019. www.eclipse.org/cdo
[8] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. 2007. A metamodel independent approach to difference representation. *Journal of Object Technology* 6, 9 (2007), 165–185.
[9] EMF Compare. last accessed February 2019. https://www.eclipse.org/emf/compare/
[10] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. 2007. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.* 376, 3 (2007), 139–163.
[11] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer.
[12] EMFStore. last accessed February 2019. www.eclipse.org/emfstore
[13] J. García, O. Diaz, and M. Azanza. 2013. Model transformation co-evolution: A semi-automatic approach. In *SLE (LNCS)*, Vol. 7745. Springer, 144–163.
[14] M. Gleicher. 2018. Considerations for visualizing comparison. *Transactions on Visualization and Computer Graphics* 24, 1 (2018), 413–423.
[15] GMF. 2019. https://www.eclipse.org/gmf-tooling/. (last accessed in June 2019).
[16] T. Kehrer, U. Kelter, and G. Taentzer. 2011. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Automated Software Engineering*. IEEE Computer Society, 163–172.
[17] S. Kelly, K. Lyytinen, and M. Rossi. 1996. MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In *Conference on Advanced Information Systems Engineering (LNCS)*, Vol. 1080. Springer, 1–21.
[18] S. Kelly and J-K. Tolvanen. 2008. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley.
[19] D. Kolovos, D. Di Ruscio, A. Pierantonio, and R. Paige. 2009. Different models for model matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models*. IEEE, 1–6.
[20] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer. 2009. Explicit transformation modeling. In *MODELS 2009 Workshops (LNCS)*, Vol. 6002. Springer, 240–255.
[21] P. Langer, M. Wimmer, J. Gray, G. Kappel, and A. Vallecillo. 2012. Language-specific model versioning based on signifiers. *Journal of Object Technology* 11, 3 (2012), 4: 1–34.
[22] Y. Lin, J. Gray, and F. Jouault. 2007. DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems* 16, 4 (2007), 349–361.
[23] S. Maoz and J. O. Ringert. 2018. A framework for relating syntactic and semantic model differences. *Software & System Modeling* 17, 3 (2018), 753–777.
[24] S. Maoz, J. O. Ringert, and B. Rumpe. 2011. A manifesto for semantic model differencing. In *MODELS 2010 Workshops (LNCS)*, Vol. 6627. Springer, 194–203.
[25] A. Mehra, J. C. Grundy, and J. G. Hosking. 2005. A generic approach to supporting diagram differencing and merging for collaborative design. In *Automated Software Engineering*. ACM, 204–213.
[26] ModelCVS. last accessed February 2019. www.modelcvs.org
[27] OCL. 2014. http://www.omg.org/spec/OCL/.
[28] D. Ohst, M. Welle, and U. Kelter. 2003. Differences between versions of UML diagrams. In *ESEC/FSE*. ACM, 227–236.
[29] R. F. Paige, N. D. Matragkas, and L. M. Rose. 2016. Evolving models in Model-Driven Engineering: State-of-the-art and future challenges. *Journal of Systems and Software* 111 (2016), 272–280.
[30] A. Schipper, H. Fuhrmann, and R. von Hanxleden. 2009. Visual comparison of graphical models. In *International Conference on Engineering of Complex Computer Systems*. IEEE, 335–340.
[31] D. C. Schmidt. 2006. Guest editor's introduction: Model-driven engineering. *Computer* 39, 2 (2006), 25–31.
[32] F. Schwägerl, S. Uhrig, and B. Westfechtel. 2015. A graph-based algorithm for three-way merging of ordered collections in EMF models. *Science of Computer Programming* 113 (2015), 51–81.
[33] Sirius. last accessed February 2019. www.eclipse.org/sirius
[34] M. Stephan and J. R. Cordy. 2013. A survey of model comparison approaches and applications. In *MODELSWARD*. SciTePress, 265–277.
[35] E. Syriani and H. Vangheluwe. 2013. A modular timed graph transformation language for simulation-based design. *Software & System Modeling* 12, 2 (2013), 387–414.
[36] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. 2013. AToMPM: A web-based modeling environment. In *Companion proceedings (MODELS'13)*, Vol. 1115. CEUR-WS.org, 21–25.

[37] M. Tanhaei, J. Habibi, and S-H. Mirian-Hosseinabadi. 2016. Automating feature model refactoring: A model transformation approach. *Information and Softw. Tech.* 80 (2016), 138–157.

[38] M. van den Brand, Z. Protić, and T. Verhoeff. 2010. Generic Tool for Visualization of Model Differences. In *International Workshop on Model Comparison in Practice*. ACM, 66–75.

[39] S. D. Vermolen, G. Wachsmuth, and E. Visser. 2012. Reconstructing complex metamodel evolution. In *SLE (LNCS)*, Vol. 6940. Springer, 201–221.

[40] V. Viyović, M. Maksimović, and B. Perisić. 2014. Sirius: A rapid development of DSM graphical editor. In *International Conference on Intelligent Engineering Systems*. 233–238.

[41] S. Wenzel. 2008. Scalable visualization of model differences. In *Workshop on Comparison and versioning of software models*. ACM, 41–46.