From two-way to three-way: domain-specific model differencing and conflict detection

Manouchehr Zadahmad Jafarlou*, Eugene Syriani*, and Omar Alam‡
*Université de Montréal, Canada
‡Trent University, Canada

ABSTRACT In collaborative work, developers evolve their models in parallel, leading to substantial differences and conflicts. To better consolidate these changes, developers need to understand the differences in terms of syntax and semantics of the models. Despite myriad efforts, the existing version control systems and model comparison tools focus on the generic models, are hardly adaptable to a domain-specific language (DSL), and primarily present syntactical changes to the developer. Furthermore, they report differences and conflicts of domain-specific models based on their abstract syntax instead of the concrete syntax of the DSL. To address these issues, we previously introduced DSMCompare to detect fine-grained and semantic differences between pairs of model versions and present the changes in the concrete syntax of the DSL. In this paper, we have further enhanced our practice by considering a three-way model comparison, typical in the context of version control systems. DSMCompare can now report differences coming from either version as well as conflicts. To detect semantic differences and conflicts, our approach relies on the DSL engineer specifying semantic differencing patterns in an editor adapted to the DSL. To evaluate DSMCompare, we reverse-engineered the commit history of several open-source projects where Java-based code refactoring changes occur. We show that DSMCompare effectively finds these semantic differences and conflicts with high accuracy.

KEYWORDS Model-Driven Engineering, Model versioning, Model differencing, Graphical concrete syntax.

1. Introduction

In model-driven engineering (MDE) projects, models are considered essential building blocks. Developers utilize domain-specific languages (DSL) to create models of the system (Kelly & Tolvanen 2008). Throughout the collaborative development process, multiple developers may modify the models (David et al. 2021). To manage these changes, MDE developers rely on version control systems (VCS) to store the models in a repository, track the change history, manage simultaneous changes, and view the differences between different model versions. Although some VCS have been designed specifically for models (Kramler et al. 2006; Altmanninger et al. 2008; Brosch et al. 2010; Koegel & Helming 2010), most practitioners use text-based VCS like Git and SVN. However, these VCS are not ideal for visualizing the differences in the history of a model in a way that is easily understandable Zadahmad et al. (2019). Generic model-based differencing tools, such as EMFCompare (Brun & Pierantonio 2008), provide differences results for classes, attributes, and association changes. But, these results are presented in the abstract syntax of the DSL, which may not be familiar to DSL users. Additionally, for large models with many elements, the fine-grained differences presented can be overwhelming for DSL users who can not track the semantics of the changes (Zadahmad et al. 2022). Therefore, there is a need for model-based differencing tools that can present difference results in a way that is more user-friendly for DSL users.

Previously, Zadahmad et al. (2019) introduced DSMCompare to address the aforementioned issues. It presents all differences between two model versions in terms of their concrete

JOT reference format:
Manouchehr Zadahmad Jafarlou, Eugene Syriani, and Omar Alam. *From two-way to three-way: domain-specific model differencing and conflict detection*. Journal of Object Technology. Vol. 22, No. 1, 2023. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2023.22.1.a5

syntax. Additionally, it aggregates fine-grained differences into semantically meaningful coarser differences expressed in the DSL semantics. However, DSMCompare uses two-way differencing, which is inadequate for many usage scenarios of VCS. When branching is used extensively, committing to the main branch must consider at least three versions of the model: the previous common version from the master branch, any version from already committed branches, and the version of the current branch. A two-way differencing tool produces different model differences based on which version is used as the base, making it inappropriate for collaborative modeling settings using VCS. For example, suppose a developer deleted an element in one version, and no change occurred in another. In this situation, the two-way differencing tool cannot determine whether a developer added an element or another developer deleted it. The answer will depend on which version is used as the base model.

Three-way differencing and merging are fundamental techniques in modern VCS (Altmanninger et al. 2009). Changes introduced concurrently by two versions must be merged using a common ancestor version. This is achieved by identifying the differences between two versions by comparing them with their common ancestor, which produces correct difference sets. The VCS reports changes specific to each version and conflicts where both versions have modified the same parts. After the conflicts have been resolved, the changes are merged into a single new version. This paper presents a novel approach for three-way domain-specific model differencing and conflict detection, which can be applied to existing two-way model differencing tool. In particular, we implemented this approach in DSMCompare (Zadahmad et al. 2019, 2022). Although n-way differencing (Owhadi-Kareshk et al. 2019; Leßenich et al. 2018) is also possible, we focus on three-way differencing because it occurs more frequently in practice. However, we discuss the possible adaptations needed to support n-way differencing.

To summarize, the contributions of this paper include:

- We provide a comprehensive solution for moving from a two-way to three-way model differencing.
- We provide a comprehensive conflict detection mechanism that can identify both fine-grained and semantic conflicts, such as equivalent changes and contradicting conflicts that may arise from three-way differencing.
- We enable DSL engineers to define semantic differencing rules for handling conflicts.
- To aid users in understanding three-way differences and conflicts, we offer visualization support with a graphical concrete syntax.
- Our approach is implemented in an EMF-based tool, which we openly share, along with a dataset of 288 Ecore models annotated with semantic differences and conflicts.

The rest of this paper is structured as follows. In Section 2, we provide an overview of the approach and introduce a running example. In Section 3, we present the necessary features for a three-way domain-specific differencing tool and justify our use of DSMCompare by comparing existing tools. In Section 4, we explain how we extend existing two-way differencing tools to the three-way approach and discuss how it generates fine-grained three-way diffs. Section 5 discusses how rules are used to semantically lift the fined-grained differences. We also explain how the concrete syntax of the original DSL can be leveraged to present the difference model in Section 6. Section 7 and Section 8 present the different types of conflicts and the algorithms we use to identify them in the difference model. In Section 9, we evaluate the effectiveness of our approach on the commit history of many open-source projects. Finally, we discuss related work in Section 10 and conclude the paper in Section 11.

2. Running example

We illustrate three-way domain-specific model differencing using the following running example. In the context of an e-commerce company, business experts, Alice and Bob, are tasked with defining the process of purchase orders and payments. To formally model this process and ensure crucial properties, such as process completeness, they model workflows using Workflow Nets (WN) (van der Aalst 1998). WN is a particular class of Petri Nets where there is a single source place, a single sink place, and all transitions are on a path from the source to the sink. In WN, transitions represent workflow tasks, places represent their pre/postconditions, and tokens represent the resources used in the workflow. The V0 model in Figure 1 shows an example of a WN model.

Charlie, the DSL engineer of the company, has built an Eclipse-based graphical modeling editor for WN that she generated from an Ecore metamodel of the DSL and a graphical concrete syntax using Sirius (Viyović et al. 2014). Figure 2 shows the Ecore metamodel of the WN DSL used in the company. She also mounted the editor with EGit (Eclipse EGit 2023) to enable the business experts to collaborate with a Git version control system installed in Eclipse. Since Alice and Bob can work simultaneously on the same WN model, they are likely to encounter conflicts when integrating their work together. However, EGit reports differences and conflicts at the XMI level of models in terms of abstract syntax concepts. As Alice and Bob are not software engineers acquainted with these concepts, Charlie wishes to offer them a domain-specific model comparison tool.

2.1. Customizing DSMCompare for WN

Unlike tools like EMF Compare (Brun & Pierantonio 2008) and EMF DiffMerge (EMF DiffMerge 2023), DSMCompare (Zadahmad et al. 2022) is an Eclipse-based tool that reports differences using the same concrete syntax as the original DSL. Furthermore, it can report more coarse-grained differences (i.e., semantic differences) and hide fine-grained differences that are irrelevant to the business experts. However, up to now, DSMCompare only supported two-way differencing, which is not suitable for most collaboration scenarios like the example above. Therefore, we continue the running example with the new DSMCompare that is presented in this paper.

To integrate DSMCompare in the WN editor, Charlie provides as input the Ecore metamodel and the `odesign` representa-

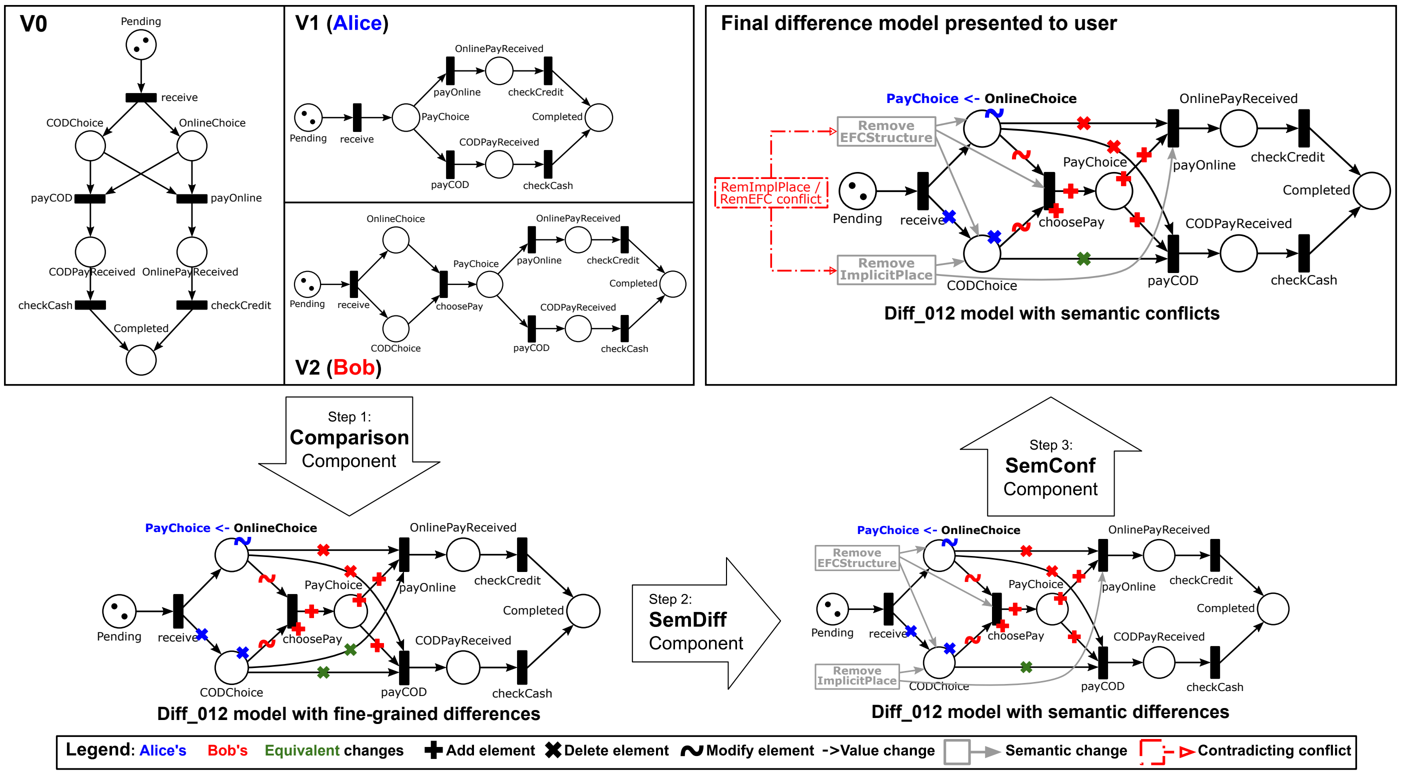

V0
Pending
receive
CODChoice
OnlineChoice
payCOD
payOnline
CODPayReceived
OnlinePayReceived
checkCash
checkCredit
Completed
V1 (Alice)
PayChoice
V2 (Bob)
choosePay
Final difference model presented to user
PayChoice <- OnlineChoice
Remove EFCStructure
Remove ImplicitPlace
RemImplPlace / RemEFC conflict
Diff_012 model with semantic conflicts
Step 1: Comparison Component
Step 2: SemDiff Component
Step 3: SemConf Component
Diff_012 model with fine-grained differences
Diff_012 model with semantic differences
Legend: Alice's Bob's Equivalent changes
Add element
Delete element
Modify element
->Value change
Semantic change
Contradicting conflict

Figure 1 Domain-specific three-way comparison of WN with DSMCompare

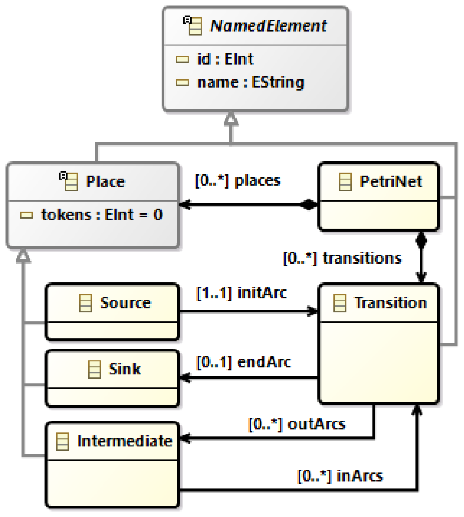
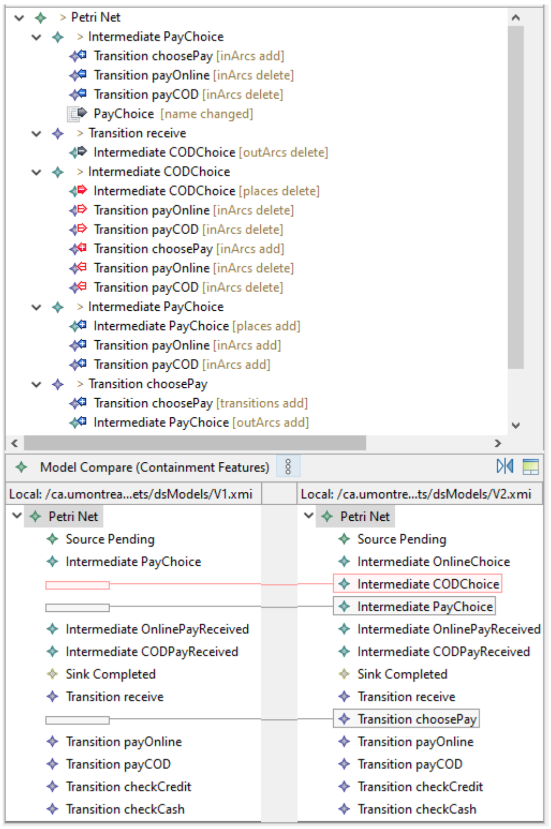


Figure 2 Metamodel of the Workflow WN DSL

tion description of the concrete syntax of WN into DSMCompare. Then, DSMCompare automatically generates a domain-specific model comparison tool tailored to compare WN models, we will call WNCompare.

WNCompare offers two editors. One editor is used by Alice and Bob to present and edit the differences between WN models. To visually present differences, WNCompare provides three default icons to each concrete syntax representation, annotated with a + / × / ∼ symbol to represent additions, deletions, and modifications, respectively. Additionally, WNCompare offers a default concrete syntax to depict semantic differences, which pertains to editing semantics (i.e., we do not map to the semantic domain as in Maoz et al. (2011)). Charlie can modify these graphics as she sees fit for WN.

The second editor in WNCompare enables Charlie to design the semantic difference rules specific to WN models. She designs three known refactoring patterns for WN, formalized in Toyoshima et al. (2015). They improve the execution of the WN by removing redundant elements in a way that does not change the observable behavior of the net. The *Remove Implicit Place* pattern removes a place connected to two transitions if there is already a path between these transitions. In the *Remove extended free-choice (EFC) structures* pattern, if a set of places are all connected to the same set of transitions, we introduce an intermediate transition and place to direct the flow from the set of places to the set of transitions. The third pattern is called *Remove TP-cross structures* where, if a set of transitions are all connected to the same set of places, we introduce an intermediate place and transition to direct the flow from the set of



```
> Petri Net
  > Intermediate PayChoice
      Transition choosePay [inArcs add]
      Transition payOnline [inArcs delete]
      Transition payCOD [inArcs delete]
      PayChoice [name changed]
  > Transition receive
      Intermediate CODChoice [outArcs delete]
  > Intermediate CODChoice
      Intermediate CODChoice [places delete]
      Transition payOnline [inArcs delete]
      Transition payCOD [inArcs delete]
      Transition choosePay [inArcs add]
      Transition payOnline [inArcs delete]
      Transition payCOD [inArcs delete]
  > Intermediate PayChoice
      Intermediate PayChoice [places add]
      Transition payOnline [inArcs add]
      Transition payCOD [inArcs add]
  > Transition choosePay
      Transition choosePay [transitions add]
      Intermediate PayChoice [outArcs add]

Model Compare (Containment Features)

Local: /ca.umontrea...ets/dsModels/V1.xmi     Local: /ca.umontre...ts/dsModels/V2.xmi
Petri Net                                     Petri Net
  Source Pending                                Source Pending
  Intermediate PayChoice                        Intermediate OnlineChoice
                                                Intermediate CODChoice
                                                Intermediate PayChoice
  Intermediate OnlinePayReceived                Intermediate OnlinePayReceived
  Intermediate CODPayReceived                   Intermediate CODPayReceived
  Sink Completed                                Sink Completed
  Transition receive                            Transition receive
                                                Transition choosePay
  Transition payOnline                          Transition payOnline
  Transition payCOD                             Transition payCOD
  Transition checkCredit                        Transition checkCredit
  Transition checkCash                          Transition checkCash
```

Figure 3 Differences and conflicts reported by EMFCompare for the running example

transitions to the set of places. Due to the complexity of these three refactoring patterns, Toyoshima et al. (2015) provide an algorithm to execute them in that order.

The top right model in Figure 1 shows a difference model produced by WNCompare, reporting fine-grained differences, semantic differences, and semantic conflicts. The semantic differences and conflicts they indicate refer directly to these refactoring patterns.

2.2. Collaboration scenario

Powered with WNCompare, Alice and Bob can use the WN editor and collaborate. Figure 4 shows their development timeline, working on their respective branch forked from the master branch using EGit. Assume the initial version in the master branch is the V0 model in Figure 1. This WN model models the payment process for a delivery service (this is a simplification of the example used in Toyoshima et al. (2015)). There are two pending deliveries represented by the tokens in the `Pending` place. When the customer receives the package, he can either pay online or cash at delivery (COD). Each payment option has its specific checkpoints before the delivery is completed. The structure of this WN ensures that each delivery is paid by only one method.

Alice and Bob branch from this version and build versions V1 and V2, respectively shown at the top of Figure 1. Alice refactors V0 by removing the implicit place `CODChoice` and renames the place `OnlineChoice` to `PayChoice`. Her change simplifies the WN model by reducing the number of places and arcs, while still ensuring a mutually exclusive payment method. In the meantime, Bob refactors V0 by removing the EFC of the places for online and COD choices, and introduces the intermediate `choosePay` transition and `PayChoice` place. While his changes also ensure the mutual exclusive property, it reduces the coupling of the places modeling the choice. Alice is the first to push her work to EGit and requests a merge. EGit accepts the request and merges her model to the master branch because there has not been any change to base model V0 since Alice branched out. Now, the head in EGit is Alice's version and points to version V1. Later, Bob finishes his work and requests a push to the master branch. EGit rejects this merge request because Bob's model is incompatible with the latest version of the base model since Alice has already changed it. To handle this issue, Bob pulls the latest version from the VCS, i.e., requests V1 to his local machine. Bob then needs a three-way differencing engine to understand where and why a conflict has occurred. More specifically, they are in conflict because Alice has removed the `CODChoice` place while Bob changed its outgoing arc.

At this point, suppose Bob used EMFCompare, a common model-based three-way comparison tool, instead of DSMCompare to understand the differences between his version V2, the common ancestor V0, and Alice's version V1. He would be presented with fine-grained differences and conflicts, as shown in Figure 3. As an expert in WN, he would have had a hard time making sense of abstract details such as "`inArcs delete`", "`outArcs add`", "`places add`", or "`transitions add`". To him, these are implementation details of the tool that lack meaning.

Now suppose Bob uses WNCompare. He selects the three model versions (his, Alice's, and their common ancestor) and launches WNCompare. After processing the differences, WNCompare presents the final difference model $Diff_{012}$ in the editor. As shown at the last step of Figure 1, the differences are expressed using the original concrete syntax of WN. It also outlines the two semantic changes stating that Alice removed the implicit place `CODChoice` and that Bob removed the EFC structure by introducing the `choosePay` transition. Thanks to WNCompare, Bob can now comprehend the reason for this conflict at the same level of abstraction as the WN models, a familiar level to Bob. Then, he can reconcile the conflict by himself or discussing it with Alice. Note that this paper only focuses on the detection and representation of differences and conflicts to DSL users in a domain-specific way, not their reconciliation and merge.

3. Features for three-way domain-specific model differencing

The comparison stage produces a list of differences and similarities by comparing two or three versions of the same artifact. There are two approaches for model comparison including operation-based and state-based comparison (Brosch, Langer, et al. 2012). Operation-based comparison relies on specific tools to edit and collect the changes from the user. In contrast, state-based operation does not restrict users to any specific tools

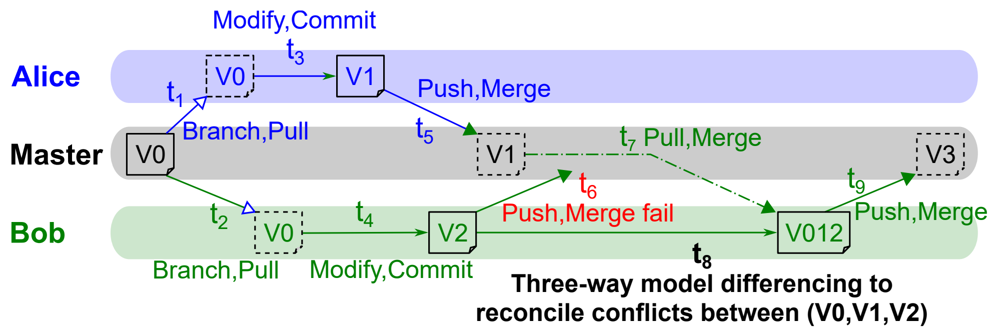


Figure 4 Collaboration scenario where three-way differencing is needed

to manipulate the models and does not need any module to collect the changes. In practice, state-based comparison is more popular and, thus, the focus of this paper. There are several state-based model comparison engines able to process models from any DSL (Stephan & Cordy 2013), such as EMFCompare (EMF Compare last accessed January 2021), DiffMerge (EMF DiffMerge 2023), DSMCompare (Zadahmad et al. 2022), Maudeling (Rivera & Vallecillo 2008), DSMDiff (Lin et al. 2007), and Epsilon-based Three-way Merging Process (E3MP) which uses the Epsilon Comparison Language (Sharbaf & Zamani 2020).

The two-way comparison in state-based approaches includes a matching step and a diffing step. The matching step finds matching elements between the two model versions. Matching identical elements can be done via identifiers, similarity algorithms, or other heuristics. The diffing step uses the matched elements to find the differences between the two models.

Three-way differencing usually combines the results of two pairwise two-way differencing of each model with their common ancestor. Rubin & Chechik (2013) and Schultheiß et al. (2021) introduce novel approaches that are faster than the pairwise method, though Schultheiß et al. (2021) preferred a pairwise comparison over a straight rating of entire matches to evaluate almost correct matches better than completely incorrect matches.

In what follows, we list the requirements for a domain-specific three-way comparison tool. We discuss to what extent existing model comparison tools satisfy these requirements.

The first three requirements are necessary for domain-specific comparison. Meeting these requirements allows for the systematic use of DSLs to represent the various aspects of a domain-specific comparison tool. As a result, it allows the tool to use DSLs to support higher-level model comparison abstractions than generic fine-grained differencing.

Meta-model agnostic. The tool can compare models conforming to any metamodel. This is necessary for the domain-specific difference so that models in any DSL can be compared.

Concrete syntax. The tool presents differences in the concrete syntax of the DSL. This improves the user experience and allows users to understand differences in the notations they are accustomed to while using the DSL.

User-defined semantics. The tool provides features to define differences meaningful to the DSL. Semantic differences are often defined in terms of rule patterns by the DSL engineer. This feature requires proper management of semantic rules, such as an editor and the possibility to infer them. This is necessary for semantic difference rule management so that new semantic rule patterns can be defined and existing rules can be updated by a DSL engineer.

The next five requirements are specific to three-way comparison. They enable the tool to detect fine-grained and semantic differences and conflicts.

Fine-grained difference detection. The tool can detect fine-grained differences based on the abstract syntax of the models provided. This is a fundamental requirement for any comparison tool. These differences capture editing changes, such as additions, deletions, modifications or rerouting of associations.

Semantic difference detection. The tool can detect semantic differences based on a predefined semantic rules for the DSL. A semantic difference is a coarsed-grained difference that groups fine-grained differences satisfying certain conditions. Semantic differences enhance the DSL user understanding when the changes are reported. This feature exposes meaningful differences in terms of the DSL. It also reduces the verbosity of the reported changes by hiding the fine-grained differences encapsulated in a semantic difference.

Equivalent change detection. The tool can detect the same changes performed in different model versions. This is applicable to fine-grained and semantic differences. It prevents duplicating changes and reduces the number of elements to ultimately merge. This feature is exclusive to three-way differencing.

Fine-grained conflict detection. The tool can detect conflicts between contradicting fine-grained changes in different model versions. Ultimately, conflicts will have to be resolved to create a valid merged model. This feature is exclusive to three-way differencing.

Semantic conflict detection. The tool can detect conflicts between contradicting semantic differences in different model versions. This features allows to reason beyond the abstract syntax of the model and facilitate the work of the DSL user when reconciling the conflicts.

Table 1 Model comparison tools satisfying the requirements: ● satisfied, ◒ partially satisfied, ○ not satisfied, ⊛ satisfied in this paper.

Requirement / Tool	DSMDiff	Maudeling	DiffMerge	E3MP	EMFCompare	DSMCompare
Meta-model agnostic	●	●	●	●	●	●
Concrete syntax	○	○	○	○	○	●
User-defined semantics	○	○	○	○	◒	●
Fine-grained difference detection	●	●	●	●	●	●
Semantic difference detection	○	○	○	○	◒	●
Fine-grained equivalent change detection	○	○	●	●	●	⊛
Fine-grained conflict detection	○	○	●	●	●	⊛
Semantic conflict detection	○	○	○	●	○	⊛
Explicit difference presentation	●	●	●	●	●	●
Headless API	○	●	●	●	●	●
Two-way differencing	●	●	●	○	●	●
Three-way differencing	○	○	●	●	●	⊛

Finally, the last four requirements focus on the degree of interaction with the model comparison tool. They enable the tool to present the differences, the DSL user to interact with the report, and to be integrated with other related tools.

Explicit difference presentation. The tool explicitly presents the results of the comparison, including the differences and conflicts. The differences can be represented in a dedicated data structure, a distinct model, or with traces showing matchings and differences. DSL users can query and visualize the results. Some tools only show the excerpt of the model involved in the differences, while others show the whole model.

Headless API. The tool can be used interactively with the user or through API. This makes the comparison tool accessible via an API for display on any device. It enabled its integrated with other tools, such as VCS or enable extensions.

Two-way differencing. The tool can compute two-way comparison and produce differences. This feature allows the DSL user to use the tool when comparing two models.

Three-way differencing. The tool can compute three-way comparison and produce differences and conflicts. This feature allows DSL users to use the tool when they are collaborating together, like in Section 2. Ultimately, the conflicts can be reconciled and the differences merged into a single valid model.

Table 1 shows to what extent different comparison tools support each requirement of a model comparison tool. All the tools we consider in the table are meta-model agnostic and can thus be used for any DSL. DSMCompare is the only tool supporting the reuse of the DSL's concrete syntax when presenting differences. EMFCompare provides extension points to manually program coarse-grained domain-specific differences (EMFCompare 2023). The scope of the newly added difference types is limited to the functionalities that existing difference types provide. E3MP does not support semantic difference detection and only focuses on conflict detection (Sharbaf & Zamani 2020). However, by default, it only outputs a list of fine-grained matches and differences using Epsilon Comparison Language (Kolovos 2009). To describe conflict detection patterns, the user must define them in Epsilon-based scripts, such as the Epsilon Validation Language (EVL), Epsilon Pattern Language (EPL), or Epsilon Object Language (EOL). They are imperative languages that combine object-oriented programming and OCL constraints. Also, for EMFCompare, updating the semantic rules can be problematic since it is not tailored to the DSL's syntax. In contrast, DSMCompare supports this feature by providing a generated domain-specific editor to define new semantic difference rules.

With no surprise, all tools can detect the fine-grained differences. DSMCompare can detect semantic differences that are defined as rule pattern models tailored to the DSL. EMFCompare support semantic difference detection but requires to program the rules in Java. But it needs expert software engineers to develop the domain-specific semantic rules. Maudeling, DSMDiff, and DSMCompare do not support three-way differencing; thus, they cannot detect equivalent and conflicting changes. E3MP only supports three-way differencing and does not focus on two-way differencing, although the underlying Epsilon Compare Language supports it. DiffMerge, and EMFCompare can detect equivalent and conflicting fine-grained differences.

All the tools represent the comparison results (differences or conflicts when applicable) as an explicit fine-grained difference model. Moreover, DSMCompare also represents the semantic differences in the model. EMFCompare models semantic differences as a sub-category of fine-grained differences. E3MP only creates a report for semantic conflicts but does not model semantic differences explicitly. The other tools do not model semantic differences. Finally, all the tools, except DSMDiff, offer API that may be used interactively with the user or other tools.

From this comparison, we deduce that DiffMerge, EMFCompare, and E3MP already support three-way differencing, while DSMDiff, Maudeling, and DSMCompare could be extended to three-way differencing. However, only EMFCompare partially support semantic differences.

Since DSMCompare is the only tool to already fully support domain-specific differencing, we choose this tool to demonstrate how to turn a two-way differencing tool into a three-way differencing tool. The process can be applied on any other tool listed, but it would require to make it domain-specific in the first place.

4. Three-way differencing support in DSMCompare

Our implementation of `DSMCompare` relies on `EMFCompare` to detect fine-grained differences and conflicts. We could have chosen another model differcing tool as long as it follows a certain API. The following list shows the minimum features that such a tool must provide to be plugged in `DSMCompare`:

Match list is composed of a set of pairs of identical fine-grained model elements from the two versions. Their identification is determined via unique identifiers or similarity heuristics.

Diff list. is composed of a set of differences between a pair of elements in the match list. The difference is regarding attribute or reference value changes.

Equivalent diff list is composed of a set of pairs from the diff list where the two elements have made the same modifications, showing an equivalent user intention.

Conflict list is composed of a set of pairs from the diff list where the changes in both elements have a contradicting user intention.

In the rest of this section, we explain how we adapt DSMCompare to support three-way domain-specific model differencing. We illustrate it with WNCompare from the running example presented in Section 2.

4.1. Generating a three-way differencing metamodel

First, we outline the generation of the difference metamodel in the context of two-way differencing. Then, we present the new extension to three-way differencing. Finally, we discuss how to handle semantic differences and conflicts.

4.1.1. Generated metamodel for two-way differencing

DSMCompare supports two-way differencing by creating a new metamodel `DSDiffMM` from the original metamodel of the

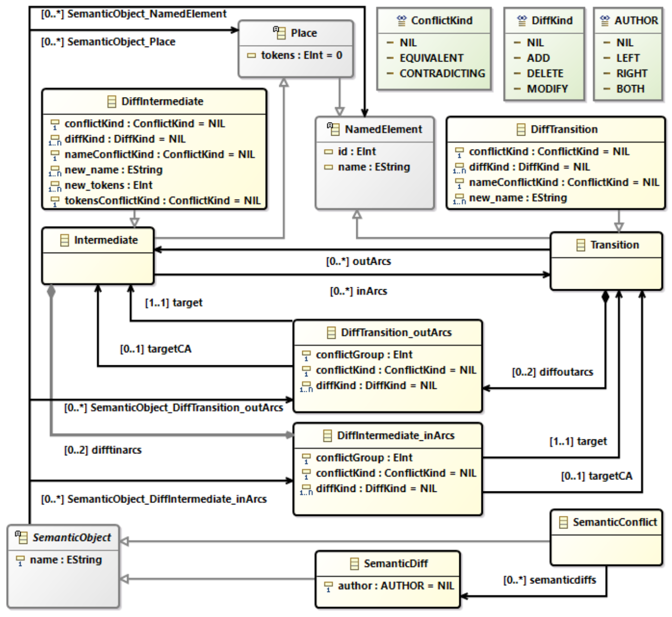


Figure 5 An excerpt of three-way Domain-Specific Difference metamodel (DSDiffMM_3Way)

DSL `MM`. We refer to $Diff_{12}$ as an instance of `DSDiffMM` that contains the results of domain-specific two-way differencing. The approach begins by creating a clone of `MM` that inherits all the DSL structural features. We then add new structural features that allow us to perform two-way differencing. In particular, for each meta-class C of `MM`, we create a corresponding difference class *DiffC* in `DSDiffMM` that extends it with *ADD*, *DELETE*, or *MODIFY* values to denote changes for objects. Each attribute of C is duplicated in *DiffC* to hold the new value in the case of a `MODIFY`. Each association *Asc* between meta-classes $C1$ and $C2$ in `MM` is refined into an intermediate class *DiffAsc* to hold *ADD* and *DELETE* values to denote changes for links.

4.1.2. Generated metamodel for three-way differencing

To transition from two-way to three-way differencing, we expand the original metamodel `MM` in the DSL to incorporate semantic changes and conflicts in a three-way manner. The idea is to reuse the structural features and style from the original DSL to remain in the spirit of the DSL. A related technique is the ramification of metamodels for domain-specific model transformations (Kühne et al. 2009). This technique emphasizes the importance of aligning meta-models with the specific domain, facilitating effective model transformations within that context. By maintaining fidelity to the original metamodel, we ensure compatibility and coherence with the underlying model, enabling meaningful representation of model differences in a domain-specific way. This involves creating a new metamodel, `DSDiffMM_3Way`, which replaces the `DSDiffMM` used for two-way differencing. The `DSDiffMM_3Way` metamodel allows us to capture specific data that is unique to three-way differencing, such as authorship information, changes in single- and multi-valued attributes and associations, fine-grained and semantic differences, as well as fine-grained and semantic conflicts, including the types of conflicts.

The process of generating the three-way DSDiffMM involves duplicating all the classes in the MM metamodel, similar to generating the two-way version, and adding new classes that enable three-way differencing and conflict detection. We refer to this resulting difference model as $Diff_{012}$. To illustrate this, we use the WN metamodel MM shown in Figure 2. Figure 5 shows an excerpt of the generated DSDiffMM_3Way, which includes some classes that DSDiffMM_3Way adds to MM. The DSDiffMM_3Way metamodel creates a detailed comparison model between three versions of the original model: V1, V2, and the common ancestor. Changes made by V1 and V2 to the common ancestor are stored in new *DiffC* classes. These classes indicate the type of modification made by each version using an array of DiffKind values. In a three-way comparison, the size of the array is 2, with indices ranging from *LEFT* to *RIGHT*. An index with a *NIL* value indicates that the corresponding version did not modify that particular element. Each *DiffC* class is also assigned a ConflictKind value, which identifies the type of conflict.

In the two-way process, DSMCompare needed to use an intermediate class called *DiffAsc* to represent each original association *Asc*. In the WN example, the DSDiffMM_3Way metamodel in Figure 5 inserts an intermediate class called DiffTransition_outArcs for the outArcs association from a transition to an intermediate place. However, if this intermediate class only points to the target of outArcs, we cannot track situations where both versions move the association to different targets. Therefore, we add a new outgoing association called *targetCA* to keep a record of the original target in the common ancestor.

The *target* outgoing association of *DiffAsc* records the change of one version. If both versions modify the target, then two instances of *DiffAsc* would be present in the $Diff_{012}$, as indicated by the [0..2] cardinality of diffoutarcs in Figure 5. To extend this to n-way differencing, we would need to change the cardinality to [0..n].

The same DiffKind enumeration is used for these intermediate classes, where a *MODIFY* value indicates that the target of the association has been modified, i.e., a *move* change as reported in EMFCompare. It is important to note that unlike class-level conflicts, an association can be conflicting with another association and is not bound within a single object. Therefore, we extend *DiffAsc* with a ConflictGroup attribute, where all *DiffAsc* instances having the same conflict group value are conflicting with each other.

The new attributes generated for two-way differencing also need to be adapted. The new generated attributes *new_A* from the original attributes *A* are now typed as an array of the type of *A*. The array size is 2 for three-way differencing (dynamic for n-way), indexed from *LEFT* to *RIGHT* order. For each attribute *A*, we also add a new attribute *A + 'ConflictKind'* to track if the value modified in V1 and V2 results in a conflict. In addition, one of the improvements we made is the ability to trace changes to multi-valued items, i.e., arrays of values or associations.

4.1.3. Tracking conflicts and provenance

In three-way differencing, we must consider that differences come from different versions and authors. This raises different situations depicted by the new ConflictKind enumeration with three values, as shown in Figure 5. *CONTRADICTING* changes indicate that two versions made different changes to the same element. Therefore, the changes on the two versions contradict each other and both changes cannot be applied. *EQUIVALENT* changes indicate that two versions made similar change edits to the same element. Therefore, we need to link them together and mark them as equivalent. *NIL* indicates that only one version made a change to an element.

We define an Author enumeration to keep track of the version accountable for each change. The values *LEFT* and *RIGHT* each of the two versions, say V1 and V2 respectively. We use the value *BOTH* to indicate that both versions applied the same change. *NIL* is reserved for an initial and default value.

Extending to n-way differencing would require replacing this enumeration with a key-value dictionary added to each class in DSDiffMM_3Way. In this case, the key represents the class or attribute name, and the value holds a list of integers. We assign each version to a positive number. If the change comes from a single version, the list contains its unique number. The list enumerates all the corresponding numbers if it comes from two or more versions.

4.1.4. Identifying semantic differences and conflicts

Three-way differencing requires a proper representation of conflicts. In DSDiffMM_3Way, we revise the SemanticDiff class to have a provenance (the Author enumeration), a meaningful description (the name attribute) of the differences, and keep track of any object it involves. The newly added SemanticConflict class in Figure 5 must also keep track of any semantic difference since conflicts can occur between fine-grained and semantic differences, like after step 3 of Figure 1. Each SemanticDiff object represents a coarse-grained difference that groups fine-grained differences into a singular difference that is domain-specific. The object is created by applying its corresponding semantic difference rule on the $Diff_{012}$ model. The SemanticConflict object represents a contradicting conflict between V1 and V2 involving at least one semantic difference.

4.2. Three-way comparison

As shown in Figure 1, we organize DSMCompare into three components. Here, we discuss the Comparison component (Step 1 in Figure 1). This component relies on the output that EMFCompare produces. It calls the three-way difference API by providing the V0, V1, and V2 models.

The Comparison component builds the $Diff_{012}$ model by creating the diff classes and associations, and setting the new values of attributes by querying the output of EMFCompare. Like EMFCompare, the $Diff_{012}$ model contains only the elements subject to change and their context. However, it should be noted that the output of EMFCompare primarily focuses on providing information regarding conflicts between fine-grained differences. Thus, we developed a four-stage procedure in the Comparison component.

First, we group the fine-grained differences produced by EMFCompare using similarity factors. For example, all changes to attributes of the same class are grouped together. The same goes for associations. Second, we collect additional information for each group. For example, we gather the attribute values shared between V1 and V2, such as the class type and original value from V0. For multi-valued attributes, EMFCompare returns one difference for each item modified in the list. Thus, we aggregate all the changes reported for that attribute and divide them into two lists: one for each version. At this point, we can already compare the values between V0, V1, and V2 and determine if the modification of the attribute is equivalent.

Third, we calculate the inter- and intra-dependencies of each group to determine the order in which DSMCompare will create the elements in $Diff_{012}$. For example, if both an attribute and its class change, DSMCompare must first create the corresponding diff class before setting its attributes. Thus, we set the attribute to depend on its class. If a class is deleted in one version, its outgoing associations are also. Therefore, associations depend on their source object within their group. Associations with the same `conflictGroup` value are also grouped together. We then sort each group according to the number of their dependencies in ascending order.

Lastly, we transform the fine-grained difference groups into the relevant instances of `DSMDiffMM` elements, and construct the $Diff_{012}$ model. DSMCompare transforms one group at a time instead of processing each difference individually to consider the changes from all versions in a single diff object, thus reducing verbosity. The process starts by creating a diff element (class diff or association diff). Then, we copy the associations and the values of all attributes from the common ancestor to the diff element. Finally, according to the type of difference (class, association, or attribute) and the kind of change (add, delete, or modify) applied to the element in each version, we set the `diffKind` and `conflictKind` attributes according to Section 4.1.3.

5. Semantic differencing

First, we define what semantic differences are. Then, we outline the specification of the semantic difference rules in the context of two-way differencing. We then present the new extension to three-way semantic differencing We also outline how semantic rules can be generated automatically.

5.1. Semantic differences

In DSMCompare, a semantic difference is defined as $SD = \langle Meaning, Constraints, Context, Filters \rangle$. *Constraints* is a set of constraints over a list of fine-grained differences. For example, Figure 7a shows the constraints defined as a graph as well as a condition that the graph pattern must satisfy. *Filters* is a list of fine-grained differences that are present in *Constraints*. It is used to hide the fine-grained differences that are encapsulated in the semantic difference. *Context* is a list of fine-grained differences that are present in *Constraints*. They are the fine-grained differences related to the semantic difference after lifting to provide a context to the semantic difference meaning. *Meaning* is a string expressed in the vocabulary of the semantic of the DSL and the editing semantics of the *Constraints*. It is the interpretation of semantically-lifting the fine-grained differences.

We implement semantic differences in DSMCompare in three steps.

5.1.1. *Aggregation of fine-grained differences*

Fundamentally, computing the difference between models investigates syntactic changes of what has been added, deleted, or modified. As we have shown in (Zadahmad et al. 2022), the difference report tends to be very verbose in the case of domain-specific models. Therefore, one way to simplify the differences reported is to encapsulate them into a coarse-grained difference that groups related fine-grained differences. This relation between a coarse-grained difference and its fine-grained differences is specified in a semantic difference rule. For example, Figure 7a illustrates the "Remove Implicit Place" SDRule. The aggregation of fine-grained differences is stated in the pattern of the semantic difference rule as a set of constraints (c.f. the *Constraints* component of the definition of *SD*). In this example, a transition (labeled 1) must have at least two outgoing arcs to two places (labeled 3 and 4). One of these places (labeled 4) must be an implicit place: there is already a path from the other place to its outgoing transition (depicted in the constraint). The implicit place and its adjacent arcs must have been deleted in one version. In conclusion, the aggregation step encapsulated fine-grained differences.

5.1.2. *Hiding verbose differences*

The encapsulated fine-grained differences can be hidden from the user to reduce the verbosity of the reported differences. However, not all fine-grained differences should be hidden to help the user understand the semantic difference with additional context. In the example, only the arc labeled 5 is filtered, while the deleted arc and place (labeled 2 and 4) are persisted.

5.1.3. *Assigning a meaning and a context*

The name of the semantic difference is essential to be meaningful in terms of the editing semantics of the aggregated fine-grained differences. Typically, the name represents common patterns in the DSL, such as refactoring or behavioral patterns. In the example, the concept of "implicit place" is not part of the syntax of WPN but of the semantics of a refactoring pattern. Moreover, a context is needed to understand which place plays the role of the implicit place and which transition ends the common path in the model. Thus, the semantic difference is associated with the place labeled 3 and the transition labeled 6.

According to Jackson & Ladd (1994), a semantic difference must use the vocabulary of the semantics, not the syntax: it must relate to the behavior of the changes. In DSMCompare, the name of the semantic difference must appropriately refer to the meaning of the changes (the *Meaning* component fulfills that). They also state that referring to a slice of the syntactic change is useful. In DSMCompare, the *Context* component fulfills that. They also argue that it must be automatically identified by the tool, which DSMCompare does by applying transformations automatically on the $Diff_{012}$ model.

5.2. Two-way semantic difference rules

In two-way, DSMCompare requires a set of domain-specific difference rules called `SDRule`. Each SDRule creates a semantic difference pattern to lift the fine-grained differences in $Diff_{12}$ semantically. To define these rules, we create a semantic differencing rule metamodel called `SDRuleMM`, along with its concrete syntax and editor. We generate a new editor for the DSL engineer to define SDRules based on a DSL for semantic differencing rules. This DSL consists of a metamodel *SDRuleMM* that is automatically generated from the *DSDiffMM*, and a concrete syntax *SDRuleCS* that is automatically generated from the *DSDiffCS*. Each class *C* in `DSDiffMM` corresponds to a pattern class *Pattern_C* in `SDRuleMM`, which includes additional attributes to uniquely identify objects, filter differences, and support negative patterns. The `SDRule` has a root class called `Rule`, which contains a constraint to restrict the applicability of a pattern rule based on attribute value changes. It also includes semantic difference objects that can refer to elements in `DSDiffMM` to encapsulate semantic differences.

Using graph-based model transformation, we transform `SDRules` into semantically equivalent Henshin rules (Strüber et al. 2017). These rules create semantic difference objects, delete objects with a `filter` attribute set to `true`, and preserve the rest of the pattern to be matched in the $Diff_{12}$ model. The `SDRule` constraint is also converted to Henshin conditions.

As multiple `SDRules` may be applicable simultaneously, DSMCompare uses a heuristic-based algorithm to schedule their application order. The priority order aims to reduce the verbosity of the presented differences and maximize the presence of semantic differences over fine-grained differences. Finally, the resulting Henshin transformation is executed on the fine-grained $Diff_{12}$ model to detect semantic differences.

We represent three-way semantic differences similarly to the previous two-way method. In three-way, the SemDiff component additionally generates a DSL to specify semantic differencing rules (*SDRule*) and applies them to the $Diff_{012}$ model.

5.3. Synthesis of three-way semantic difference rules

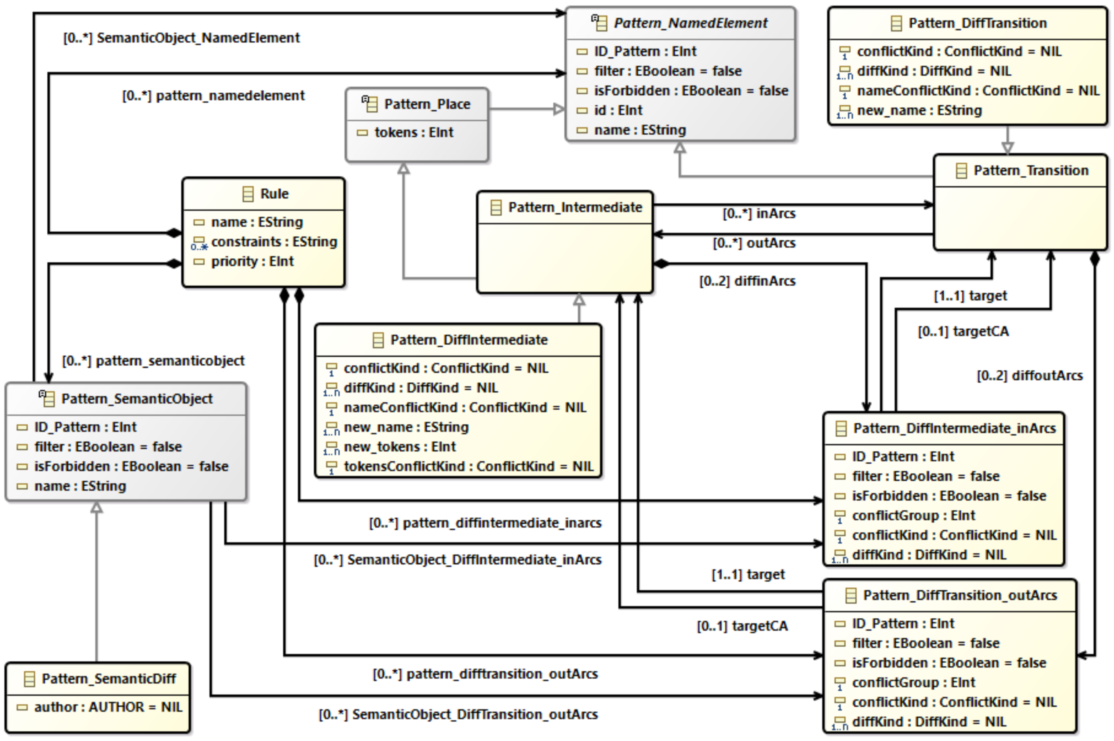
The `SemDiff` component automatically generates the rule metamodel `SDRuleMM` from the `DSDiffMM` metamodel. Figure 6 shows a fragment of the result for the WN example, which extends the process outlined in Section 5.2. One improvement in the generation process is that the `Rule` root class of the `SDRuleMM` metamodel can now contain multiple instances of the root class of the DSL (`Pattern_PetriNet` in our example) in case more than one version modified it. It can also contain an instance of any other class to keep the patterns as compact as possible. One particularity for the three-way *SDRules* is that they should define patterns over the fine-grained differences pertaining to the same author.

To apply the *SDRules*, we transform them into Henshin graph transformation rules that can then be applied to the $Diff_{012}$ model. Figure 7a shows the *Remove Implicit Place* rule that is defined using the automatically generated SDRule concrete syntax and editor for the WN domain in WNCompare, along with its equivalent rule in Henshin Figure 7b. The graph transformation rule modifies the $Diff_{012}$ model to show semantic differences and filters unnecessary fine-grained differences. This rule matches a *Transition* object conneted to a *Intermediate (place)* object labeled *n3*, with *outArcs* association from one side, and conneted to a *DiffTransition_outArcs* object with *diffoutArcs* association from another side. The rule makes sure that, *DiffTransition_outArcs* object is connected to *DiffIntermediate* object, the *DiffIntermediate* object is connected to *DiffIntermediate_inArcs* object, and *DiffIntermediate_inArcs* object is connected to the final *Transition* object labeled *n6*. The rule, also calls *pathExistBtw* method by passing *n3 and n6* parameters, as an additional constraint, to check if there is an alternative path between *n3 and n6* transitions.

When the rule finds this pattern in the $Diff_{012}$ model, it creates a *SemanticDiff* object named "Remove Implicit Place" associated with Intermediate and Transition objects. Note here that, since the current version of Henshin does not support attributes of an array type, we generate two variables (one for left and one for right) to split the values of the array for three-way differencing.

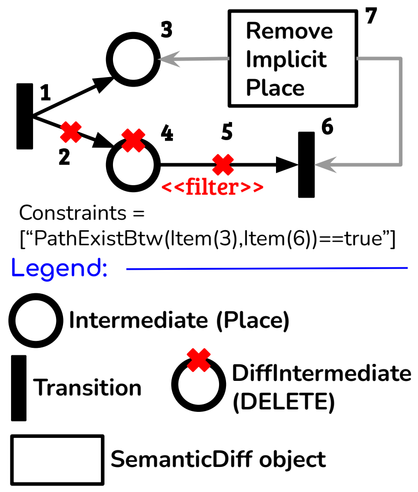
We now outline the transformation processes to generate a Henshin graph transformation rule *HRule* from an *SDRule*. As an example, we use the *Remove Implicit Place* rule depicted in Figure 7.

1. Create an HRule with the same name as the SDRule.
2. Create a node in HRule with the action *preserve* for every pattern object in SDRule that has no *filter* and no *NAC_group* (e.g., node `n5` in the example).
3. Create a node with the action *delete* in HRule for every pattern object with *filter* set to true in SDRule.
4. Create a node with the 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*.
5. Create a node with the action *create* in HRule for each *SemanticDiff* object in SDRule (e.g., node `n7`).
6. Create a condition in HRule with the *OR* operand that duplicates the conditions defined in SDRule for both left and right versions. For example, `n4diff_kind` is separated into `n4diff_kind_Left` and `n4diff_kind_Right` for node `n4` to cover both versions.
7. Create an edge with action *create* in HRule for each association adjacent to a *SemanticDiff* node in SDRule (e.g., `SemanticObject_NamedElement` between nodes `n7` and `n3`).
8. Create an edge with action *delete* in HRule for each association adjacent to a pattern object with *filter* attribute set to a true in SDRule (e.g., `diffinArcs` between nodes `n4` and `n5`).
9. Create an edge with action *forbid* in HRule for each association adjacent to a pattern object with *NAC_group* attribute set to a value in SDRule.
10. Create an edge with action *preserve* in HRule for each association adjacent to a pattern object with *NAC_group* and *filter* attributes not set to a value in SDRule.

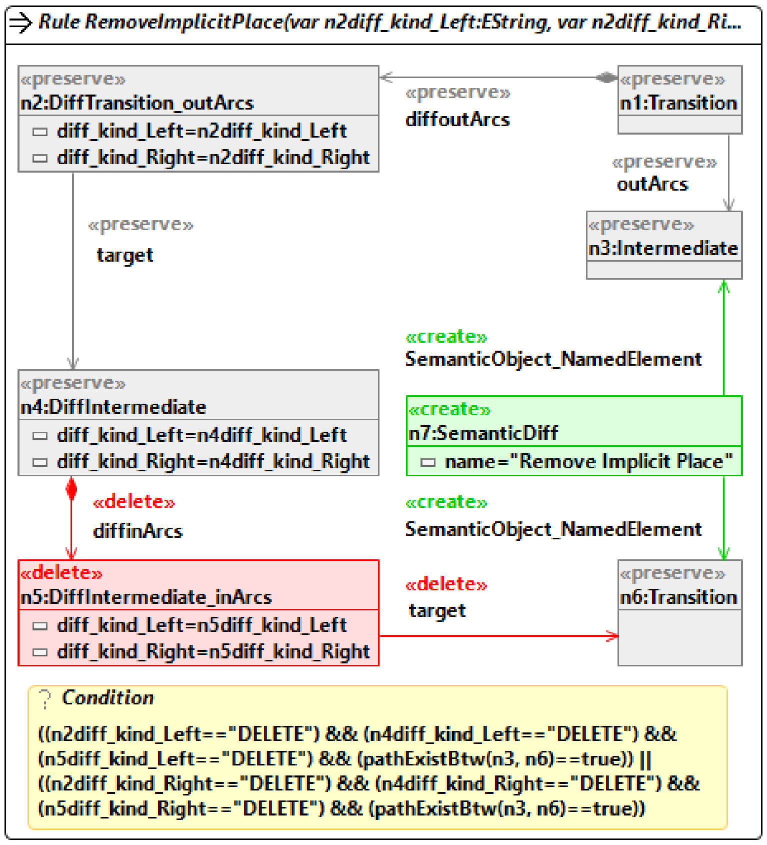


Pattern_NamedElement
ID_Pattern : EInt
filter : EBoolean = false
isForbidden : EBoolean = false
id : EInt
name : EString
Pattern_DiffTransition
conflictKind : ConflictKind = NIL
diffKind : DiffKind = NIL
nameConflictKind : ConflictKind = NIL
new_name : EString
[0..*] SemanticObject_NamedElement
[0..*] pattern_namedelement
Pattern_Place
tokens : EInt
Pattern_Transition
Rule
name : EString
constraints : EString
priority : EInt
Pattern_Intermediate
[0..*] inArcs
[0..*] outArcs
[0..2] diffinArcs
[1..1] target
[0..1] targetCA
[0..2] diffoutArcs
Pattern_DiffIntermediate
conflictKind : ConflictKind = NIL
diffKind : DiffKind = NIL
nameConflictKind : ConflictKind = NIL
new_name : EString
new_tokens : EInt
tokensConflictKind : ConflictKind = NIL
[0..*] pattern_semanticobject
Pattern_SemanticObject
ID_Pattern : EInt
filter : EBoolean = false
isForbidden : EBoolean = false
name : EString
Pattern_DiffIntermediate_inArcs
ID_Pattern : EInt
filter : EBoolean = false
isForbidden : EBoolean = false
conflictGroup : EInt
conflictKind : ConflictKind = NIL
diffKind : DiffKind = NIL
[0..*] pattern_diffintermediate_inarcs
[0..*] SemanticObject_DiffIntermediate_inArcs
[1..1] target
Pattern_DiffTransition_outArcs
ID_Pattern : EInt
filter : EBoolean = false
isForbidden : EBoolean = false
conflictGroup : EInt
conflictKind : ConflictKind = NIL
diffKind : DiffKind = NIL
[0..1] targetCA
[0..*] pattern_difftransition_outArcs
[0..*] SemanticObject_DiffTransition_outArcs
Pattern_SemanticDiff
author : AUTHOR = NIL

Figure 6 A fragment of the semantic differencing rules metamodel



(a) The"Remove Implicit Place" semantic differencing rule



Rule RemoveImplicitPlace(var n2diff_kind_Left:EString, var n2diff_kind_Ri...
«preserve»
n2:DiffTransition_outArcs
diff_kind_Left=n2diff_kind_Left
diff_kind_Right=n2diff_kind_Right
«preserve»
diffoutArcs
«preserve»
n1:Transition
«preserve»
outArcs
«preserve»
target
«preserve»
n3:Intermediate
«create»
SemanticObject_NamedElement
«preserve»
n4:DiffIntermediate
diff_kind_Left=n4diff_kind_Left
diff_kind_Right=n4diff_kind_Right
«create»
n7:SemanticDiff
name="Remove Implicit Place"
«delete»
diffinArcs
«create»
SemanticObject_NamedElement
«delete»
n5:DiffIntermediate_inArcs
diff_kind_Left=n5diff_kind_Left
diff_kind_Right=n5diff_kind_Right
«delete»
target
«preserve»
n6:Transition
Condition

```
((n2diff_kind_Left=="DELETE") && (n4diff_kind_Left=="DELETE") &&
(n5diff_kind_Left=="DELETE") && (pathExistBtw(n3, n6)==true)) ||
((n2diff_kind_Right=="DELETE") && (n4diff_kind_Right=="DELETE") &&
(n5diff_kind_Right=="DELETE") && (pathExistBtw(n3, n6)==true))
```

(b) The "Remove Implicit Place" rule in Henshin

Figure 7 A semantic difference rule transformed into a Henshin graph transformation rule

The `SemDiff` component generates the Henshin rules from the repository of domain-specific *SDRules*. The algorithm found in Zadahmad et al. (2022) schedules their order of application. This algorithm optimizes the verbosity of the displayed fine-grained differences and emphasizes semantic differences over syntactic differences. Therefore, the final $Diff_{012}$ model output from the `SemDiff` component contains semantic differences and fine-grained differences not involved in semantic difference patterns.

5.4. Generating *SDRules* from examples

The DSMCompare graphical editor allows the `DSL` engineer to create a new *SDRule*. DSMCompare also provides a new feature to reduce the time and effort to define a semantic difference rule. When the difference between two consecutive versions shows a semantic change (operational semantic), the DSL engineer can turn it into an *SDRule*. She simply needs to provide the fragment of each model version showing the semantic change and DSMCompare creates the corresponding *SDRule* following these steps. First, it produces the $Diff_{012}$ model to get the differences between the two model versions. Then, it transforms the model into a draft of an *SDRule* model. In this step, it roughly processes every structural feature and transforms each element to its corresponding element in the `SDRuleMM`. Finally, the DSL engineer can manually set the `filters` and the `NAC_groups` in the pattern. She must also set the name, constraints, and creates the semantic difference object that the rule encapsulating. She can then test the new *SDRule* by applying its Henshin equivalent on the given $Diff_{012}$ model. Our experience has shown that this reduces the effort required to create `SDRules`.

6. Tailoring the concrete syntax of difference models and semantic rules

DSMCompare provides a concrete syntax to display the $Diff_{012}$ model. We implement this feature using Sirius (Sirius 2023), one of the most popular frameworks to generate graphical modeling environments in the Eclipse ecosystem. As explained in Section 2, the DSL engineer needs to provide a concrete syntax, *CS*, of her DSL using Sirius. However, she does not need to supply a new one for *DSDiffMM*. To maintain the DSL's soul, DSMCompare automatically creates a default version of the domain-specific difference concrete syntax, *DSDiffCS*, that reuses the style from *CS*. The foundation of Sirius' definition of concrete syntax is a viewpoint specification model, also known as `odesign`. It establishes a mapping between graphical representations and *MM* elements.

For instance, we define a `NodeMapping` in Sirius that references an icon in an image file to render the graphical representation of the `Place` class. A combination of text, icons, shapes, and style adjustments, such as size and color, can be used in the "Code Node Mapping". Similarly, an `EdgeMapping` produces associations. In terms of compositions, a `BorderedNodeMapping` is used to render the target class inside of the `NodeMapping` of the source class. Sirius uses, the Acceleo Query Language, a subset of OCL, to express the constraints that can filter graphic representations according to a condition. We also automatically generate a palette of buttons to instantiate the classes and relationships of *MM*.

6.1. Automatic synthesis of the concrete syntax

We create *DSDiffCS* using an outplace transformation that accepts *CS* as input and produces *DSDiffCS* as output. By doing so, we are able to reuse the properties and styles of similar concrete syntax elements from the *CS* within *DSDiffCS*. We implemented this transformation in ATL to help automate the process. The general logic of the transformation is to extend the representation of each related *MM* class to construct the representation of each `Diff_` class, then duplicate each component of *CS* onto *DSDiffCS*. This maximizes the usage of *CS* to represent the difference model in a way that makes sense to DSL users. For each `NodeMapping`, e.g., `PlaceNode`, we create nine diff nodes that each represent a combination of two difference kind pairs from two versions of an element such as: `DiffPlaceNodeADD_ADD`, `DiffPlaceNodeADD_DELETE`, `DiffPlaceNodeADD_MODIFY`, etc. By default, the diff node is identical to the original node marked with a pair accompanied by a symbol as shown in the legend of Figure 1.

Assume that the diff class `DiffA_S` corresponds to the association with an incoming composition `diffsS` from class `A` and outgoing associations `target` and `targetCA` to `B`. For example class `DiffTransition_outArcs` corresponds to the association with an incoming composition `diffoutarcs` from class `Transition` and outgoing associations `target` and `targetCA` to `Intermediate`. In *DSDiffCS*, `DiffA_S` is represented with a `BorderedNodeMapping` as a subnode of the `NodeMapping` of `A`, and we create `BorderedNodeMappings` for each `Edge`. *DSDiffCS* uses a `BorderedNodeMapping` to represent `DiffA_S` as a subnode of `A`. We also build `BorderedNodeMappings` for each `Edge`.

6.2. Layering the differences

The three-way difference model has more elements than in two-way, increasing the complexity of understanding it. Therefore, we implemented a layering system provided by Sirius to organize the graphical difference model elements better. Each diagram element is a member of a *Layer*. The user can enable or disable each layer to only visualize the elements of interest and decrease the verbosity of the reported differences.

We create three layers for the $Diff_{012}$ model. The first layer groups all fine-grained differences, including details such as the kinds of differences. The second layer shows all semantic differences, more specifically, the semantic difference objects and their associations. The third layer shows all conflicts between semantic and/or fine-grained differences. Associations are only visible in their specific layer; e.g., the conflicts between related objects only appear in the third layer.

6.3. Themed provenance of the differences

We also use distinct themes to distinguish between the changes made by the V0 and V1 authors. For example, we can play with the color darkness or assign a specific set of colors to distinguish between them. The user can customize the concrete syntax at will.

The editor generated to define *SDRules* also reuses *DSDiffCS*. This allows the DSL engineer to define a rule in a concrete syntax with which DSL users are familiar.

7. Detecting equivalent changes

The utilization of three-way differencing can lead to conflicts, which are addressed in this section and the subsequent one. Specifically, in this section, we delve into equivalent changes while the next section discusses contradicting conflicts. These conflicts are identified in Step 3 in Figure 1. When two versions, V1 and V2, make alterations to the same element relative to V0, it results in a conflict. An equivalent change arises when V1 and V2 make identical modifications. DSMCompare provides the necessary features to detect equivalent changes and visualize them to the DSL user by reusing the concrete syntax of the DSL.

7.1. Equivalent fine-grained conflicts

Four kinds of equivalent changes can occur for class differences. V1 and V2 add a new class instance, delete an existing object, or modify an attribute with the same value. If the attribute is multi-valued (e.g., an array), the changed values must also occur on the same index. In addition, four kinds of equivalent changes can occur for association differences. V1 and V2 add a new association instance between two objects, delete an existing link from the source object, or modify a link by redirecting it to the same the target object. If the association has a cardinality greater than one, an equivalent *MODIFY* conflict occurs if all target objects are the same in both versions.

Detecting fine-grained equivalent changes is straightforward in DSMCompare, thanks to the enumerations explained in Section 4.1. When the `DiffKind` array in an object or link has the same values (*ADD/ADD*, *DELETE/DELETE*, *MODIFY/MODIFY*) and their values are the same, it assigns the *EQUIVALENT* value to the `ConflictKind` of the element in the $Diff_{012}$ model.

The concrete syntax for equivalences is adapted automatically thanks to the *DiffMM_3way* metamodel and *DiffCS*. As depicted in Figure 1, the $Diff_{012}$ model only displays one symbol for the equivalent change using a predefined color for equivalence (green in this case).

7.2. Equivalent semantic changes

An equivalent semantic change occurs when both versions accomplish identical semantic differences. For example, consider the case both of left and right authors working on a WN model apply a *Remove Implicit Place* semantic change on an identical place. In both versions, a place is connected to two transitions. EMFCompare displays six fine-grained equivalent changes with three equivalent differences for each side. In DSMCompare, we show the three equivalent fine-grained conflicts: the place object, the `inArcs` link, and the `outArcs` link are deleted. After applying the `SemDiff` component, the semantic difference rule *Remove Implicit Place* is applied, which assigns a single `SemanticDiff` object with the same name.

Then, the `SemConf` component proceeds as follows. For all the association targets connected to the `SemanticDiff` object, e.g., *Remove Implicit Place*, we check the values for the `diffKind` and `ConflictKind` attributes. If in all the connected objects, the values in the `diffKind` array are all equal and the value for the `ConflictKind` is *EQUIVALENT*, we set the value of the `author` attribute (recall from Figure 5) in the `SemanticDiff` object to *BOTH*. It indicates that both versions have made equivalent semantic changes. If in all the objects connected to the `SemanticDiff` object, the first value in the `diffKind` array is not *NIL*, but the other value is *NIL*, we set the *LEFT* value for the `author` attribute. In the opposite case, we set the value of the `author` attribute to *RIGHT*. This results in a single `SemanticDiff` object for both versions and is connected to the target fine-grained difference objects for both versions.

Please note that equivalent semantic changes do not necessarily cover all fine-grained/fine-grained equivalent differences because some fine-grained differences may not contribute to any semantic difference.

8. Detecting contradicting conflicts

A contradicting conflict occurs when V1 and V2 have made different changes to the same element with respect to V0. Fine-grained contradicting conflicts arise when the `diffKind` array of a specific element has either *MODIFY/MODIFY* or *MODIFY/DELETE* values. A fine-grained change may also conflict with a fragment of a semantic change. Semantic conflicts can result in a semantically unacceptable model, meaning that the model is syntatically valid but violates some semantics of the domain. For example, in Figure 1, a *Remove Implicit Place* semantic difference is detected in V1 and V2 deleted the source transition in the semantic difference: the two changes are syntactically valid. A naive reconciliation of these differences would merge the two versions by applying the fine-grained differences of *Remove Implicit Place* and removing the source transition. However, semantically, they are contradicting because either the source transition should deleted or the modifications related to *Remove Implicit Place* should be applied. Therefore, DSMCompare detects contradicting conflicts for fine-grained/fine-grained, fine-grained/semantic, and semantic/semantic differences. It also visualizes these conflicts to the DSL user.

8.1. Fine-grained conflicts

As explained in Section 4.2, we divide fine-grained differences into similarity groups, aggregate the required properties, and process each group separately. After processing each difference group, we set the value of the `conflictKind` attribute to *CONTRADICTING* if the values in the `diffKind` array have different values other than *NIL*.

The first column in Figure 8 demonstrates a *MODIFY/DELETE* contradicting conflict on objects, where V1 changed the token value of a place, while V2 removed the place. Visually, DSMCompare labels the contradictions with user-specific colors to quickly distinguish them. It also sets the `ConflictKind` of the place object to *CONTRADICTING*.

The second column in Figure 8 demonstrates a *MODIFY/DELETE* contradicting conflict on links. Here, V1 rerouted the arc outgoing from the `Pending` place to the `payOnline` transition, while V2 removed this arc. EMFCompare reports that V1

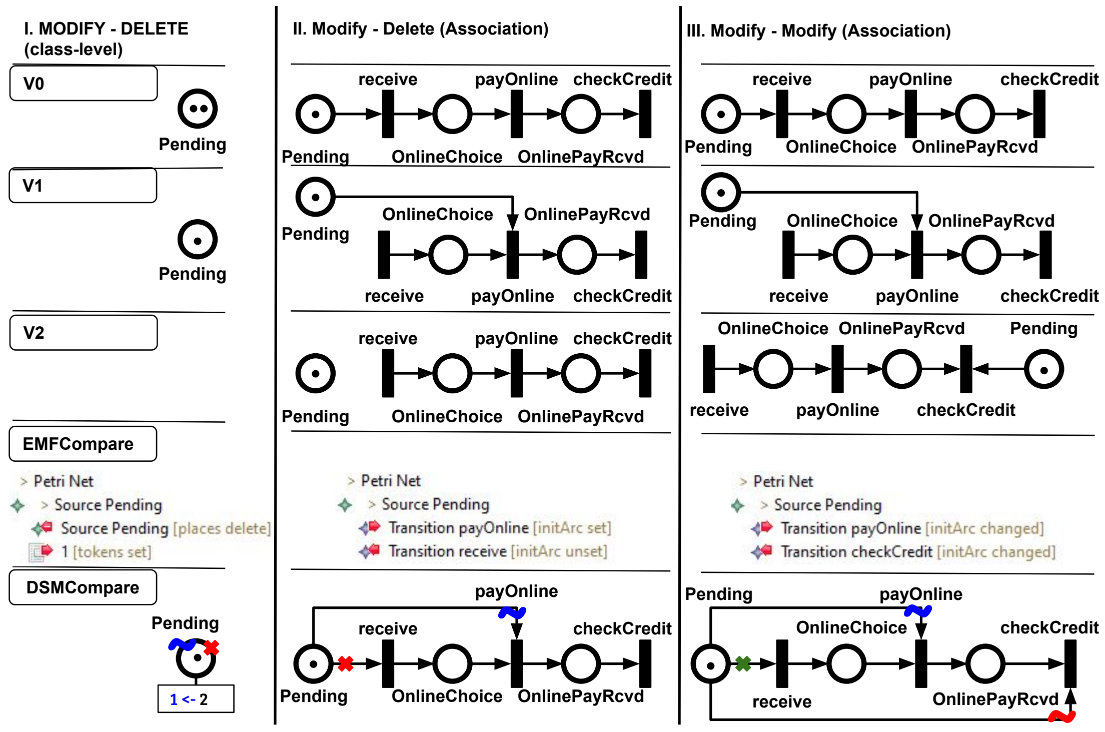

I. MODIFY - DELETE (class-level)
II. Modify - Delete (Association)
III. Modify - Modify (Association)
V0
V1
V2
EMFCompare
DSMCompare
Pending
receive
payOnline
checkCredit
OnlineChoice
OnlinePayRcvd
Petri Net
Source Pending
Source Pending [places delete]
1 [tokens set]
Transition payOnline [initArc set]
Transition receive [initArc unset]
Transition payOnline [initArc changed]
Transition checkCredit [initArc changed]
1 <- 2

Figure 8 Examples of fine-grained contradicting conflict

has unset the arc and V2 has set it. In contrast, DSMCompare post-processes this information and reports these two links as conflicting with visual cues. Additionally, it assigns the same `ConflictGroup` value (recall from Figure 5) to both diff objects representing the links.

The last column in Figure 8 shows a contradicting *MODIFY/MODIFY* conflict with the same situation as in the secund column, except that, now, V2 has also rerouted the arc to another transition `checkCredit`. EMFCompare does not report the move from the initial target link. In contrast, DSMCompare computes this information showing that the original target of the arc has changed (as an equivalent change $\times$) to different targets in each version (as two contradicting modifications $\sim$). In this case, all three links share the same `ConflictGroup` value. Note that DSMCompare treats multi-valued attributes and associations similarly for these contradicting conflicts.

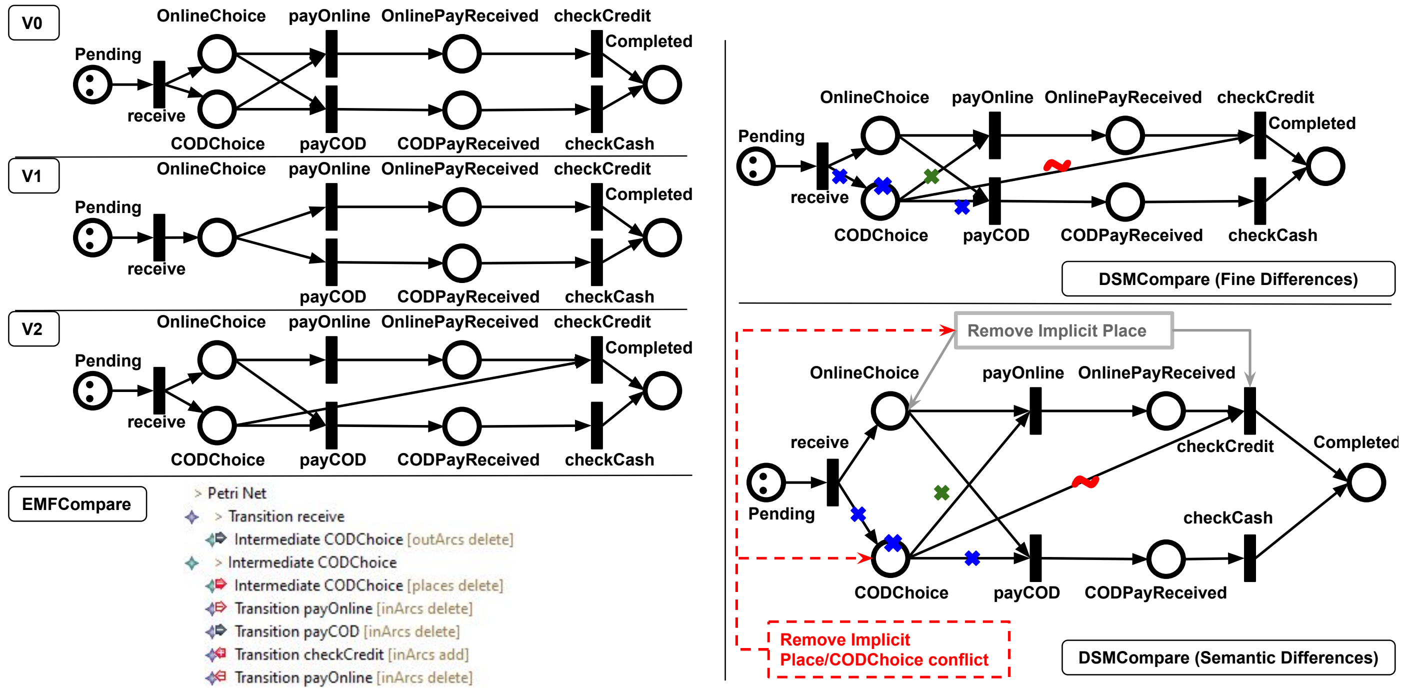
8.2. Potential conflicts between fine-semantic and semantic-semantic differences

A semantic difference typically involves changes on multiple fine-grained difference elements, encapsulating them in a common change intention. Unlike contradicting conflicts between fine-grained differences, conflicts involving semantic differences can overlap across multiple elements and must be carefully identified. For example, in Figure 9, suppose V1's changes represent that the implicit place `CODChoice` is removed (i.e., deleting the redundant place object and removing all links connected to it), while V2 only reroutes the arc of that place to a different transition `checkCredit`. Then, there is a contradicting fine-semantic conflict because of the overlap on the `inArcs` change. Furthermore, like in Figure 1, suppose that V1's changes represent removal of a place (deleted while there is another alternate path) and, in V2, there is a change of the target transition of *CODChoice* place's `inArcs` link, i.e., removing an EFC structure. Then, there is a contradicting semantic-semantic conflict between *Remove Implicit Place* and *Remove EFC structures* semantic difference objects because of the overlap on the *CODChoice* object and the `inArcs` link.

Thanks to its rule-based approach, DSMCompare can detect these situations automatically and report them to the DSL user to facilitate conflict reconciliation. Detecting overlapping conflicts can be time-consuming with respect to the number of *SDRules* available and the number of occurrences of these conflicts. Therefore, DSMCompare pre-computes the potential conflicts between fine-grained and semantic differences at design-time (i.e., when DSL engineers produce their *SDRules*); thus, it only requires computing them once.

To find the potential conflicts between semantic differences, we compute the conflicts and dependencies between *SDRules*. Since *SDRules* are transformed into Henshin graph transformation rules, we perform a critical pair analysis (CPA) (Lambers et al. 2008) and multi-granular conflict and dependency analysis (Multi-CDA) (Lambers et al. 2018) on these rules. To detect potential conflicts between fine-grained and semantic differences, we synthesize a Henshin rule for every possible `DiffKind` of the `DSDiffMM_3Way` metamodel elements. We generate a rule for adding, removing, and modifying classes and associations. We also generate a rule for every attribute modification. For example, the generated rule *DiffTransition_outArcs* encapsulates the removal of an `outArcs` association from a transition.

Henshin offers support for Multi-CDA through its API. It detects all potential conflicts between the rules, such as a rule



> Petri Net
> Transition receive
Intermediate CODChoice [outArcs delete]
> Intermediate CODChoice
Intermediate CODChoice [places delete]
Transition payOnline [inArcs delete]
Transition payCOD [inArcs delete]
Transition checkCredit [inArcs add]
Transition payOnline [inArcs delete]

Figure 9 Example of contradicting semantic-fine conflict

creating an element that another one forbids or a rule deleting an element that another one modifies. For example, it can detect the conflict between the *Remove Implicit Place* semantic rule and the place deleted fine-grained rule because the former rule *uses* the place object. However, Henshin's API for Multi-CDA does not detect attribute-level conflicts; therefore, we use its CPA feature for these situations. It can detect that a change in an attribute value, like the tokens, potentially conflicts with another rule that uses it. CPA returns the pair of rules in conflict because of the attribute change.

Multi-CDA returns a matrix in which each entry shows a value indicating the number of potential conflicts between two rules. The pair consists of either a fine-grained or a semantic rule. A non-zero value in the matrix indicates a potential conflict between the rules. We can deduce the reason for each conflict from the resulting matrix. For example, it shows a 1 for the pair *deleteIntermediate_inArcs* and *Remove Implicit Place* rules. This means that deleting the `inArcs` association from an intermediate place object to a transition object in one version has one potential conflict with the *Remove Implicit Place* rule in the other version.

For the WN DSL, we generate 28 fine-grained Henshin rules. Together with the 3 semantic difference rules, computing all the potential conflicts between 31 rules is time-consuming due to the exponential time complexity of CPA and Multi-CDA. Therefore, pre-computing them at design-time saves a significant amount of time for the DSL user exploring the conflicts.

8.3. Computing actual conflicts for fine-semantic differences

Multi-CDA and CPA indicate only potential conflicts between rules. Therefore, we need to verify if they effectively occur in the $Diff_{012}$ model.

Algorithm 1 Calculate all actual conflicts between fine-grained and semantic differences

```
Input: conflicts list of potential conflicts, diff012 difference model
 1: procedure SETCONFLICTFINESEM(conflicts, diff012)
 2:     semDiffList ← GETSEMANTICDIFFS(diff012)
 3:     for all sem in semDiffList do
 4:         potConflicts ← GETCONFLICTSFS(sem, conflicts)
 5:         actualConflictList ← ∅
 6:         for all pc in potConflicts do
 7:             fine ← FINDINSTANCEOF(sem, pc, diff012)
 8:             if fine.conflictKind = CONTRADICTING then
 9:                 actualConflictList ← actualConflictList ∪ {fine}
10:         if |actualConflictList| > 0 then
11:             semConflict ← CREATESEMCONFLICT(sem)
12:             for all fineConflict in actualConflictList do
13:                 CREATEASSOCIATION(semConflict, fineConflict)
```

Algorithm 1 shows how the `SemConf` component calculates the actual conflicts between fine-grained and semantic differences. Given the list of potential conflicts (see Section 8.2), it creates a `SemanticConflict` object. It links it to the `SemanticDiff` and fine-grained difference elements if the conflict really exists in the $Diff_{012}$ model. The algorithm starts by collecting all the `SemanticDiff` objects in the $Diff_{012}$ model. The function GETSEMANTICDIFFS returns a list of the names of all these objects corresponding to the *SDRule* that created them. For example, in Figure 9, this function returns *"Remove Implicit Place"*. Then, the algorithm searches through the list of potential conflicts to identify all fine-grained rules that conflict with each of these *SDRules* (line 4). It then verifies if these potential fine-grained rules have effectively been used in the $Diff_{012}$ model. For a given `SemanticDiff` object, the function FINDINSTANCEOF returns the fine-grained difference object that is linked to it. Again, a mapping by name corresponds fine-grained objects to their fine-grained rules. On line 9, the *actualConflictList* set stores all fine-grained difference objects that are in contradicting conflict and overlapping with a semantic difference object. In the example, the set contains `DiffIntermediate` and `DiffIntermediate_inArcs` as the contradicting conflicting fine-grained differences with the *"Remove Implicit Place"* semantic difference (*USE-DELETE* and *DELETE-MODIFY* respectively). Finally, lines 10–13 create a `SemanticConflict` object for each of these instances and link it to the semantic difference object and all fine-grained difference objects in the *actualConflictList* set.

The conflict list input to Algorithm 1 results from the Multi-CDA findings, which computes potential conflicts involving class-level or association-level changes. However, as explained in Section 8.2, we rely on CPA to compute potential conflicts involving attribute-level changes. Therefore, we devise a modified version of Algorithm 1 to determine the actual conflicts between attribute modifications and semantic differences. The main changes are on lines 7–9 to find the corresponding attribute. In this case, the function FINDINSTANCEOF searches for a *DiffClass* instance where the attribute corresponding to the fine-grained rule has been modified by both versions. It also checks that the conflict kind of the *DiffClass* instance is contradicting. Furthermore, this attribute must be involved in one of the constraints of the *SDRule*. If this situation occurs in the $Diff_{012}$ model, the algorithm adds the *DiffClass* instance to the *actualConflictList*. Like in Algorithm 1, it then creates a `SemanticConflict` object and links it to the semantic and the fine-grained difference objects. The first row in Figure 9 illustrates this situation.

8.4. Computing actual conflicts for semantic-semantic differences

Algorithm 2 Calculate all actual conflicts between semantic differences

```
Input: conflicts list of potential conflicts, diff012 difference model
 1: procedure SETCONFLICTSEMSEM(conflicts, diff012)
 2:     semDiffList ← GETSEMANTICDIFFS(diff012)
 3:     for all (sem1, sem2) in semDiffList do
 4:         potConflicts ← GETCONFLICTSSS(sem1, sem2, conflicts)
 5:         for all pc in potConflicts do
 6:             fine1 ← FINDINSTANCEOF(sem1, pc, diff012)
 7:             fine2 ← FINDINSTANCEOF(sem2, pc, diff012)
 8:             if fine1=fine2 ∧ fine1.conflictKind=CONTRADICTING then
 9:                 semConflict ← CREATESEMCONFLICT(sem1,sem2)
10:                 CREATEASSOCIATION(semConflict,sem1)
11:                 CREATEASSOCIATION(semConflict,sem2)
```

Algorithm 2 shows how the `SemConf` component calculates the actual conflicts between semantic differences. Given the list of potential conflicts (see Section 8.2), it creates a `SemanticConflict` object and links it to the `SemanticDiff` elements if the conflict really exists in the $Diff_{012}$ model. Like in Algorithm 1, it starts by collecting all the `SemanticDiff` objects in the $Diff_{012}$ model. However, now it only considers pairs of semantic-semantic conflicts (*sem1* and *sem2* on line 3). The function GETCONFLICTSSS searches through the list of potential conflicts to identify all fine-grained rules that conflict with both *sem1* and *sem2*. For example, let us consider the $Diff_{012}$ model in Figure 1. The function GETSEMANTICDIFFS returns the two *SDRules*: *Remove Implicit Place* and *Remove Efc Structures*. Both have semantic conflicts between them. Thus, the function GETCONFLICTSSS returns `DiffIntermediate` and `DiffIntermediate_inArcs` objects as the contradicting conflicting fine-grained differences between them (*USE-DELETE* and *DELETE-MODIFY*, respectively). Lines 6–7 return the actual fine-grained difference object involved in these semantic difference objects in the $Diff_{012}$ model. The algorithm then verifies that it is really the same fine-grained difference object that these semantic difference objects share. It also checks that it is in a contradicting status For the example, the algorithm actually determines that the `DiffIntermediate_inArcs` object is in a contradicting conflict state. Finally, lines 10–11 create a `SemanticConflict` object and link it to the semantic difference objects. It is shown in dashed lines in Figure 1.

9. Evaluation and discussion

We evaluate DSMCompare using model histories created by third parties. We first give implementation details on the three-way DSMCompare. Then, we present our experiment and discuss the results. We also outline some limitations of our approach.

9.1. Implementation

We implemented DSMCompare as an Eclipse plug-in that runs on the Eclipse Modeling Framework (EMF version 2022-03). The tool is downloadable through the open-source repository[1]. To find the generic model-based matches and differences, we instantiate the CDOCompare engine[2], the default EMFCompare engine in Eclipse. We rely on the API of EMFCompare to retrieve the difference set between the three versions. Using Java, the `Comparison` component transforms these generic differences into an instance of the `DSDiffMM_3Way` metamodel using the EMF API. For the `SemDiff` component, we use Xtend[3] to transform domain-specific rules into Henshin textual format. Then, using the Henshin API, we find potential conflicts among semantic and fine-grained differences (using Multi-CDA). With this information, we calculate the optimal order of execution of the rules. Then, we execute the rules to enhance the $Diff_{012}$ model with semantic differences. Finally, the `SemConf` component calls the Multi-CDA and CPA Java APIs and finds the conflicts between semantic and fine-grained differences following the algorithms presented in Section 8.

9.2. Objectives

We now present the evaluation of DSMCompare following an experiment we conducted. We already demonstrated that two-

[1] https://github.com/geodes-sms/DSMCompare/
[2] https://www.eclipse.org/cdo/ last accessed Jul 2022
[3] https://www.eclipse.org/xtend/index.html last accessed Jul 2022

way DSMCompare reduces verbosity, improves the detection of semantic differences, and is effective in practice (Zadahmad et al. 2022). Here, we evaluate the accuracy of DSMCompare in finding semantic differences and conflicts in a three-way differencing setting. We do not explicitly consider fine-grained differences because they were covered in the previous evaluation and are used to populate semantic differences and conflicts. Furthermore, we do not evaluate the graphical features of the tool that are related to visualizing conflicts, as such an evaluation would require conducting a user study, which is beyond the scope of this paper.

Therefore, in this experiment, we evaluate three-way DSMCompare with respect to the following research questions:

RQ1 Does DSMCompare correctly detect three-way semantic differences?

RQ2 Does DSMCompare correctly detect three-way semantic conflicts?

9.3. Experiment setup

We explain the process of collecting the data required for the experiment and the evaluation procedure.

9.3.1. Data collection. Due to the lack of existing repositories of domain-specific models and their associated semantic difference rules, the subject of our experiment is Java programs reverse-engineered into Ecore models. As explained in (Zadahmad et al. 2022), we consider a refactoring pattern as a semantic difference if it has been applied in a new version of a model. Similarly, we consider a refactoring pattern as a semantic conflict if it is involved in the conflict between three model versions. Furthermore, GitHub is a source of a significant number of code-based projects that may be transformed into Ecore models, allowing us to assess their histories for refactoring changes and domain-specific differences and conflicts.

Figure 10 describes the process we followed to build a dataset of labeled Ecore model versions for three-way differencing and conflict detection. The initial step is to select the GitHub repositories on which we conduct our study. GitHub, the predominant host of open-source projects, reports having over 42 million public repositories[4] in June 2022. Munaiah et al. (2017) examined GitHub repositories and offered Score-based and Random Forest classifiers to identify well-engineered software repositories. They have shown that the latter classifier has a greater accuracy rate. Therefore, we filter out projects not identified as well-engineered by the Random Forest classifier using their dataset of 1 857 423 repositories.As suggested by Pinto et al. (2018), we further assure the quality of the repositories by using the number of stars and community involvement as repository selection metrics. Thus, we only consider repositories with communities of two or more people and 500 or more stars on GitHub. Furthermore, we choose only Java-based repositories given the toolset we use. We now have a dataset of 104 repositories for our experiment. However, we discovered that nine of these repositories are not available via the GitHub URL supplied; therefore, we excluded them from the study. This leads us to a total of 95 repositories to consider.

[4] https://github.com/search?q=is:public retrieved on 12 June 2022.

To answer both research questions, we must consider repositories with merge conflicts involving object-oriented refactorings in their history. To this end, we use the *RefConfMiner* project (Shen et al. 2019), which is forked from *RefactoringsInMergeCommits* (Mahmoudi et al. 2019). It uses the *RefactoringMiner* project, a popular refactoring detection tool that currently can detect 87 distinct refactoring types in Java repositories (Tsantalis et al. 2020). The output is stored in a database containing the commit IDs of refactoring-related merge conflicts. For each commit, it also records the identifiers of the version triplet (V0, V1, V2) and the detected refactoring type. From the 95 repositories, only 13 projects include at least three refactoring-related merge conflicts that can be processed with *RefactoringMiner* and downloaded successfully. The project names are `android`, `closure-compiler`, `error-prone`, `jabref`, `junit4`, `mcMMO`, `POSA-14`, `querydsl`, `realm-java`, `redpen`, `storm`, `syncany`, and `titan`. Label A in Figure 10 marks the 96 conflicting commits produced by *RefConfMiner* that involve at least one refactoring.

From these commits, we use the output of *RefConfMiner* in *MergeScenarioMiner* (Shen et al. 2021) to collect the Java files involved in conflicting merge commits. It produces a folder for each conflicting commit ID, containing three sub-folders: the parts of the project involved in V0, the changed parts of the project in V1, and those in V2. However, the folders only contain Java files that DSMCompare is unable to manage.

Therefore, in the third step, we convert the source code to the corresponding Ecore model representations, using Eclipse's *MoDisco* framework (Bruneliere et al. 2014). The result is an instance of Knowledge Discovery Metamodel (KDM) which represents the structure and behavior of an entire software. With *MoDisco*, we transform the KDM model of the three versions (V0, V1, V2) into Ecore models.

We built a simplified MiniJava metamodel in Ecore, shown in Figure 11, to represent packages, classes, attributes with their type and cardinality, methods with their signature, and method bodies as one string. The MiniJava models represent the source code we collected in label A in Figure 10. Each MiniJava model includes all the Java files (i.e., packages, and classes) for one of the three versions involved in the merge commit. Since we have 96 merge commits, we end up with 288 MiniJava models in Ecore. Additionally, we have manually prepared 17 semantic difference rules to encapsulate the refactorings. We reused those from the experiment in (Zadahmad et al. 2022) and adapted them to our MiniJava metamodel. Figure 12 shows the *Pull-up Method* semantic differencing rule, which encodes that the method of a sub-class is moved to its super-class.

9.3.2. Methodology. We compare the collected data from *RefConfMiner* with the MiniJava models processed by DSMCompare. Given the three instances of the MiniJava metamodel for each conflicting commit, we call the API of EMFCompare and produce the three-way fine-grained generic model-based differences. Then, we pass the differences through the `Comparison` and `SemDiff` components and generate Diff_{012} models, including the semantic three-way differences shown by label B in Figure 10. Finally, we pass the Diff_{012} model produced by

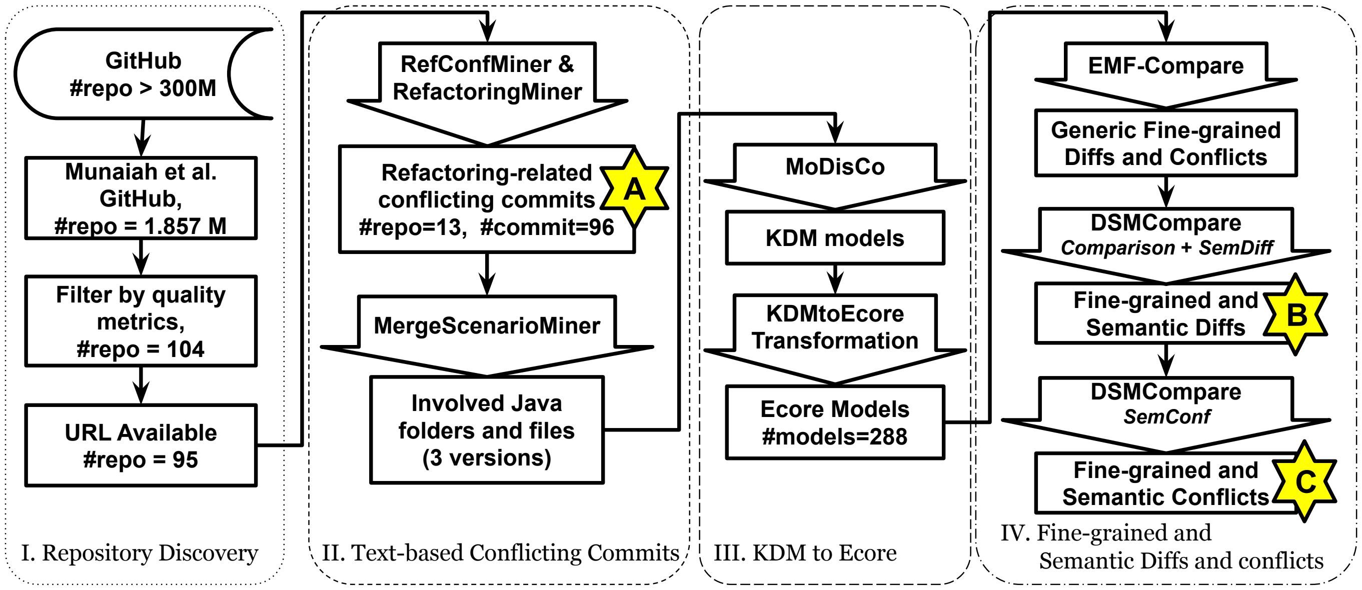

GitHub #repo > 300M
Munaiah et al. GitHub, #repo = 1.857 M
Filter by quality metrics, #repo = 104
URL Available #repo = 95
I. Repository Discovery
RefConfMiner & RefactoringMiner
Refactoring-related conflicting commits #repo=13, #commit=96
A
MergeScenarioMiner
Involved Java folders and files (3 versions)
II. Text-based Conflicting Commits
MoDisCo
KDM models
KDMtoEcore Transformation
Ecore Models #models=288
III. KDM to Ecore
EMF-Compare
Generic Fine-grained Diffs and Conflicts
DSMCompare Comparison + SemDiff
Fine-grained and Semantic Diffs
B
DSMCompare SemConf
Fine-grained and Semantic Conflicts
C
IV. Fine-grained and Semantic Diffs and conflicts

Figure 10 Evaluation setup to execute DSMCompare on Java code from GitHub repositories

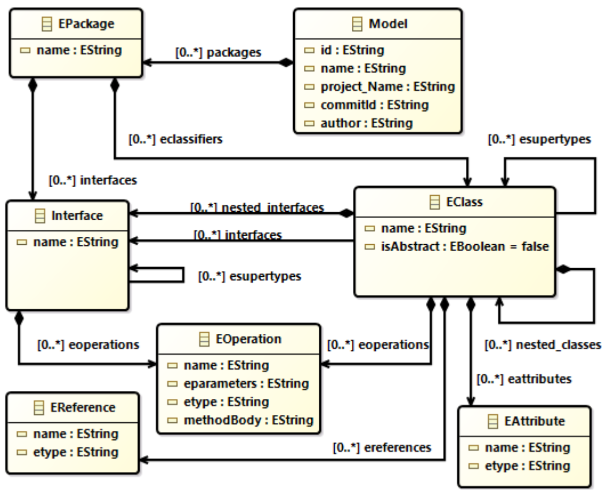


Figure 11 The *MiniJava* metamodel

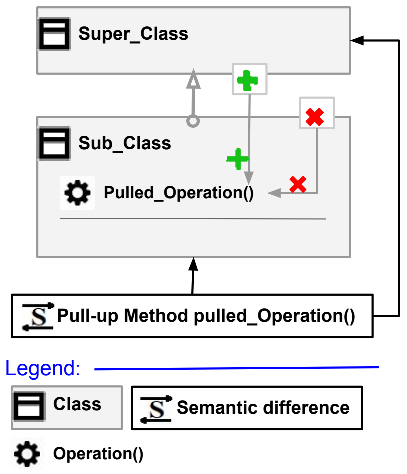


Figure 12 The *Pull-up Method* semantic differencing rule

the `SemDiff` component through the `SemConf` component and populate $Diff_{012}$ model with semantic conflicts (label C in Figure 10).

To answer *RQ1*, we compare the refactorings found by *RefConfMiner* (label A), with the three-way semantic refactoring differences reported by DSMCompare (label B). We use semantic differences and conflicts found by *RefConfMiner* as the baseline for DSMCompare. We denote the two sets AD_i and B_i for each commit i, respectively. We rely on precision and recall measures for the comparison. In Equation (1), we define precision as the ratio between the correctly found differences in DSMCompare and the total number of differences it finds. We define recall as the ratio between the correctly found differences in DSMCompare and the expected number of differences found by *RefConfMiner*.

$$\begin{aligned} Precision_{diff} &= \sum_{i=1}^{96} \frac{|AD_i| \cap |B_i|}{|B_i|} \\ Recall_{diff} &= \sum_{i=1}^{96} \frac{|AD_i| \cap |B_i|}{|AD_i|} \end{aligned} \tag{1}$$

To answer *RQ2*, we compare the conflicts found by *RefConfMiner* (label A) with the fine-grained and semantic conflicts reported by DSMCompare (label C). We denote the set AC_i of conflicts found by *RefConfMiner* for each commit i. We also rely on precision and recall measures for the comparison. In Equation (2), we define precision as the ratio between the correctly found conflicts in DSMCompare and the total number of conflicts it finds. We define recall as the ratio between the correctly found conflicts in DSMCompare and the expected number of conflicts found by *RefConfMiner*.

$$\begin{aligned} Precision_{conf} &= \sum_{i=1}^{96} \frac{|AC_i| \cap |C_i|}{|C_i|} \\ Recall_{conf} &= \sum_{i=1}^{96} \frac{|AC_i| \cap |C_i|}{|AC_i|} \end{aligned} \quad (2)$$

We manually perform compare each difference output in DSMCompare with *RefConfMiner*. For each refactoring or conflict reported by *RefConfMiner*, we analyze the report, including the Java file and the name of the involved elements (e.g., package, class, method, attribute). We then manually compare it with the results in the corresponding $Diff_{012}$ model output by DSMCompare.

9.4. Characterization of the resulting dataset

We first present some key findings in the resulting dataset produced by DSMCompare.

Table 2 Summary of the results after applying DSMCompare

Number of		
Projects		13
Commits		96
MiniJava models in Ecore		288
Semantic difference (refactoring) rules		14
Total number of		
Fine-grained differences		11 287
Fine-grained diffs involved in semantic diffs		7 342
Semantic differences		3 059
Remaining fine-grained differences		3 945
Fine-grained conflicts		657
Semantic differences involved in conflicts		474
***Diff*$_{012}$ elements per commit**		
Median	Average	Standard dev.
135	266	318
Semantic differences per commit		
Median	Average	Standard dev.
13	32	60
Fine-grained differences per commit		
Median	Average	Standard dev.
37	118	206
Semantic conflicts per commit		
Median	Average	Standard dev.
3	5	6
Fine-grained conflicts per commit		
Median	Average	Standard dev.
4	7	8

9.4.1. Description of the resulting dataset. In total, we produced 96 triplets of Ecore models representing the three versions of each commit. They are accompanied by 96 Ecore models representing the three-way differences and conflicts ($Diff_{012}$) annotated with all the refactoring operations. Out of the 87 refactoring types (Tsantalis et al. 2020), we only found occurrences of 14 semantic difference rules on these models. The complete dataset is available online[5]. Table 2 presents a summary of the results that DSMCompare has found.

The results show that DSMCompare can find a considerable amount of semantic and fine-grained differences in a large collection of projects and related conflicting commits. Note that

[5] https://doi.org/10.5281/zenodo.7386968

since we downloaded only the Java files involved in the conflicting commits for all three versions, the $Diff_{012}$ model only presents the minimal model needed to understand the context of the changes in each version, as opposed to showing the complete models. Therefore, we consider that $Diff_{012}$ models with an average size of 266 elements are quite large. In this metric, we count the different MiniJava model elements: package, class, interface, association, attribute, and method objects. We do not include the number of attributes for each element, such as the *method body* of method objects.

We also note that fine-grained differences account for around half of the $Diff_{012}$ model sizes, which means that almost half of each minimal model is changed. Semantic differences cover 65% of all fine-grained differences. Therefore, thanks to the semantic differences, the DSL user is left with only 35% of fine-grained differences to interpret and investigate. This drastic reduction in verbosity concurs with the results in (Zadahmad et al. 2022) and helps the DSL user better understand the differences. We observe that, on average, 16% of the semantic differences are in conflict (semantic-semantic and semantic-fine conflicts), while 6% of the fine-grained differences are in conflict. These ratios show that semantic conflicts do occur in practice and, in this dataset, they occur more often than fine-grained conflicts.

Figure 13 categorizes the number of refactorings that DSMCompare has found per commit per project. For each project, we sort the commits by decreasing number of semantic differences it contains. For example, project `syncany` has 14 commits ranging from 80 semantic differences in commit 1 to two in commit 12. Most refactorings are found in project `realm-java` with 432 semantic differences in a single commit. It also consistently has the most refactoring differences in nine commits. With only five commits, project `titan` is second in this order with the most refactoring differences (174) in commit 3. Project `jabref` arrives third, topping 128 refactoring differences in a single commit. Interestingly, project `closure-compiler` consistently contains an average of 21 semantic differences across its 10 commits. Project `syncany` has the most number of commits (14) with at least 31 semantic differences. Table 2 shows an average of 32 refactoring differences found in each of the 96 commits across all projects. The chart Figure 13 characterizes the vast diversity of the dataset under study.

Figure 14 shows a similar chart but for conflicting refactorings, i.e., semantic differences involved in a contradicting conflict. In this case, project `closure-compiler` has most of the conflicts with 42 in a single commit. It also has the most number of conflicts in nine commits. Interestingly, 53% of the semantic differences are involved in conflicts for this project, whereas the average across all commits of all projects is 36%. In comparison, project `realm-java` had the most refactoring differences (Figure 13), but only 20% of them are involved in conflicts. We notice fewer variations in the number of conflicts than in the number of differences across the commits, given that there are five conflicting refactorings on average per commit.

9.4.2. Example output.

Figure 15 illustrates the kind of results that DSMCompare outputs for the dataset. It shows a fragment of a $Diff_{012}$ model related to a conflicting three-way merge commit. The visualization is generated from the concrete syntax we defined in Sirius for the MiniJava DSL. In the excerpt of this difference model, DSMCompare captures the contradicting conflict in which V1 (in blue, authored by *HeartSaVioR* in GitHub) moved the method `MultiPut` from class `RedisClusterMapState` to its base class `AbstractRedisMapState`. Following the *SDRule "Pull-up Method"*, DSMCompare creates a `Pull-up Method` semantic difference object and associates it with the `MultiPut` method and the `AbstractRedisMapState` class. However, V2 (in red, authored by *ptgoetz* in GitHub) modified the body of this method. Therefore, DSMCompare recognizes this conflict between fine-grained (V2) and semantic (V1) differences and annotates the `MultiPut` method a *MODIFY/DELETE* icon (blue $\times$ and red $\sim$). In addition, it creates a contradicting semantic conflict object and associates it with *"Pull-up Method"* semantic difference and `MultiPut` method.

This particular commit includes eight refactorings that are involved in semantic conflicts. Conflicts mainly occur in two Java files: `RedisMapState.java` and `RedisClusterMapState.java`. Each file comprises one class, three nested classes, and one nested interface. The outline on the right of Figure 15 shows the complete $Diff_{012}$ model to enable the DSL user to navigate to the desired location of the model. In the bottom-right of the figure, a panel shows the different properties of the selected conflict, including its name and the associated elements.

9.4.3. Types of refactorings.

DSMCompare has found a similar distribution of the refactoring differences and conflicts as in (Mahmoudi et al. 2019). Figure 16 shows the frequency of each refactoring type across all commits of the dataset. For each type of refactoring, the chart presents the distribution of semantic differences in black and refactorings involved in contradicing conflicts (semantic-semantic and semantic-fine conflicts) in white. This figure shows that DSMCompare can find various types of semantic differences from difference patterns. For example, simple patterns, like the *Rename class* rule, identify a single attribute change. Patterns like the *Move class* rule identify associations between classes and packages. Patterns like the *Pull-up method* rule rely on the properties of methods, associations between classes and methods, and constraint checking.

The chart in Figure 16 is sorted in terms of the frequency of the semantic differences (refactorings). The most common refactoring differences found in the dataset (more than 10%) include *Extract and move method*, *Extract method*, *Move class*, and *Rename method*. Whereas the most common refactorings involved in conflicts include *Rename method*, *Extract method*, and *Extract and move method*. This is expected because renaming a method causes multiple conflicts, such as *Rename/Delete method*, *Rename/Add Method*, and *Rename/Rename method*. Similarly, extracting a method from its original class causes conflicts when another version modifies the same method. We also observe that renaming an element and modifying a property of a method generates more conflicts, whereas moving a method, class, or attribute generates fewer conflicts.

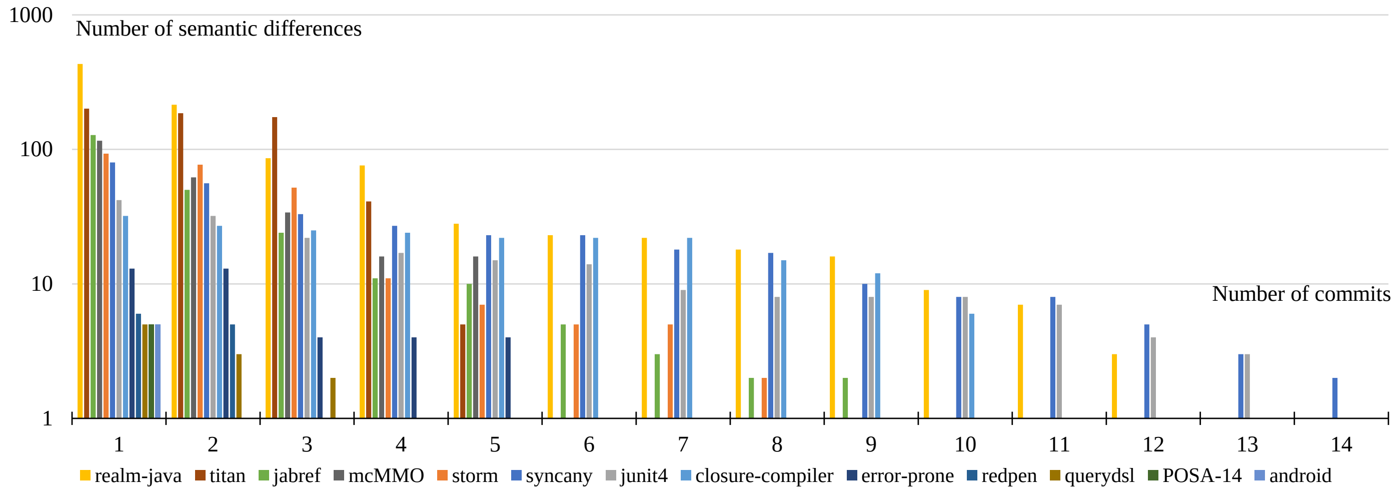


Figure 13 Number of semantic differences per commit

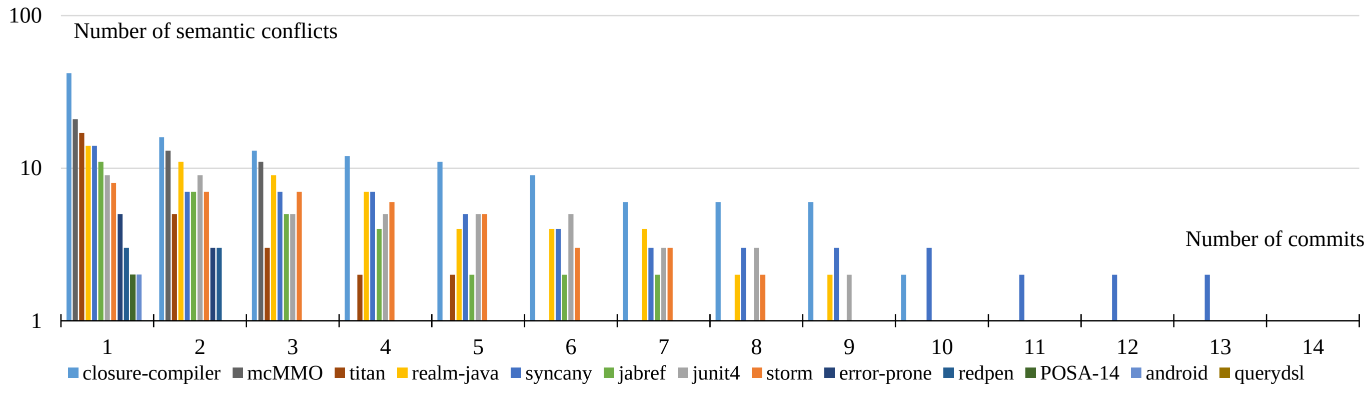


Figure 14 Number of semantic conflicts per commit

9.5. Effectiveness of DSMCompare

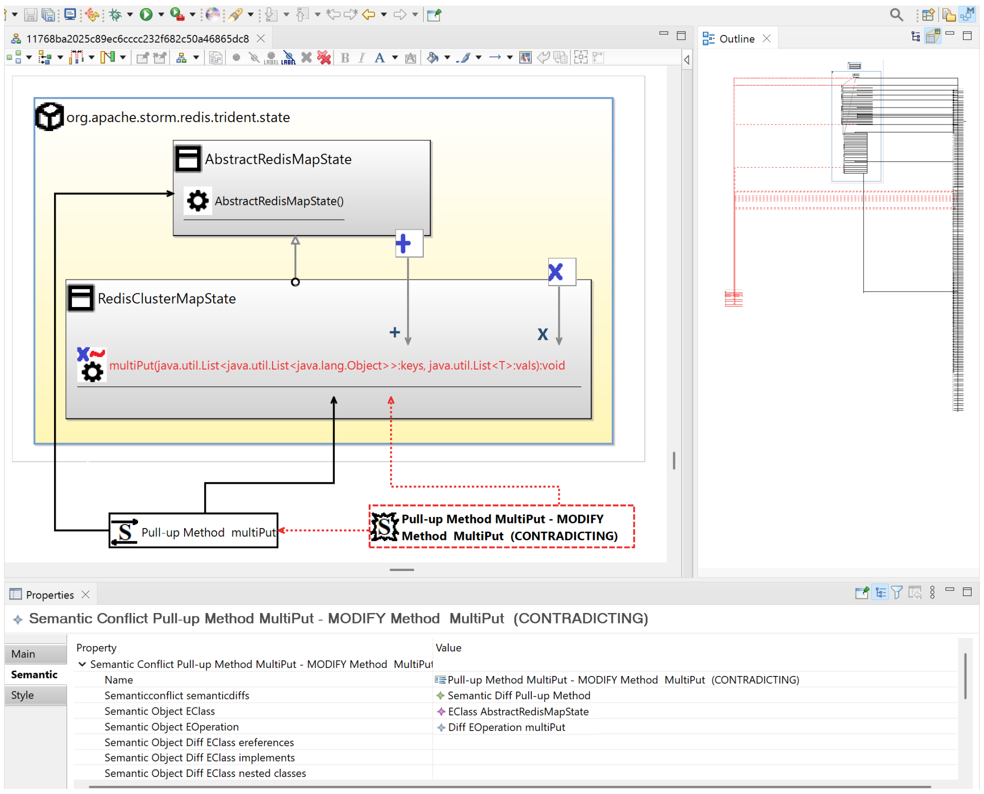
We now evaluate the results in terms of precision and recall. Figure 17 shows the overall precision and recall of refactorings differences and conflicting refactorings that DSMCompare found from the dataset. The results are calculated according to Equations (1) and (2) using *RefConfMiner* as the baseline of comparison. Recall that, in this experiment, finding semantic differences means finding differences where a refactoring type is involved. Finding semantic conflicts includes conflicts between semantic-semantic and semantic-fine differences, thus any conflict involving a refactoring.

The overall trends of all four box plots are very high, with an average of at least 96%. There is almost no variation in the precision of differences and conflicts and in the recall of conflicts: the standard deviation and variance coefficient are under 5%, and the interquartile range is 0. The near-perfect scores indicate that DSMCompare correctly found almost all the semantic differences and conflicts identified by *RefConfMiner*. Note that true negatives are not possible in this experiment since we only look for refactorings. Therefore, the accuracy of DSMCompare is also near perfect.

The only exception to these observations is the recall of the differences with a standard deviation and variance coefficient of 7.5%, and an interquartile range is 6%. Nevertheless, the average recall of differences is still very high at 96%. Some false negatives occur when DSMCompare misses one or two differences in some $Diff_{012}$ models containing very few differences. For example, in commit `2b59ffd` of project `syncany`, *RefConfMiner* finds three refactoring differences, while DSMCompare only finds two, leading to a precision and recall of 67%. In the rare situations where DSMCompare incorrectly identified a semantic difference (false positives), the refactorings are related to the body of the method, which is captured as an unstructured string in our dataset of MiniJava models. For example, the *Inline method* refactoring type transfers the content of a method to a method calling it. Most of the situations where DSMCompare missed a semantic difference (false negative) were because the files involved in the semantic differences were not available to be downloaded. Nevertheless, the F1-score for both semantic differences and conflicts is 97%, showing the high accuracy and effectiveness of DSMCompare.

9.6. Discussion

With these results, we can now answer our two research questions.


11768ba2025c89ec6cccc232f682c50a46865dc8
Outline
org.apache.storm.redis.trident.state
AbstractRedisMapState
AbstractRedisMapState()
RedisClusterMapState
multiPut(java.util.List<java.util.List<java.lang.Object>>:keys, java.util.List<T>:vals):void
Pull-up Method multiPut
Pull-up Method MultiPut - MODIFY Method MultiPut (CONTRADICTING)
Properties
Main
Semantic
Style

Semantic Conflict Pull-up Method MultiPut - MODIFY Method MultiPut (CONTRADICTING)

Property	Value
Semantic Conflict Pull-up Method MultiPut - MODIFY Method MultiPut	
Name	Pull-up Method MultiPut - MODIFY Method MultiPut (CONTRADICTING)
Semanticconflict semanticdiffs	Semantic Diff Pull-up Method
Semantic Object EClass	EClass AbstractRedisMapState
Semantic Object EOperation	Diff EOperation multiPut
Semantic Object Diff EClass ereferences	
Semantic Object Diff EClass implements	
Semantic Object Diff EClass nested classes	

Figure 15 Excerpt of the difference model showing semantic differences and conflicts in the Sirius for commit `11768ba` in the `storm` project

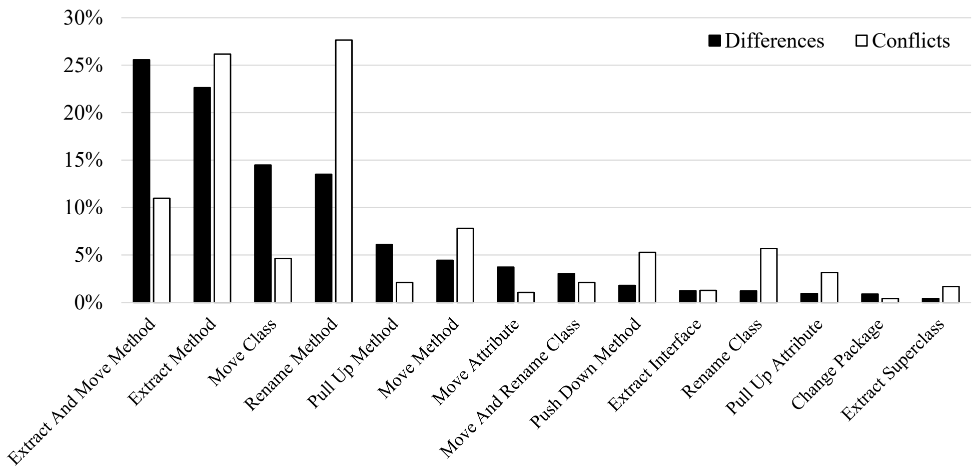


Figure 16 Frequency of refactoring types reported by DSMCompare compared to those involved in a conflict

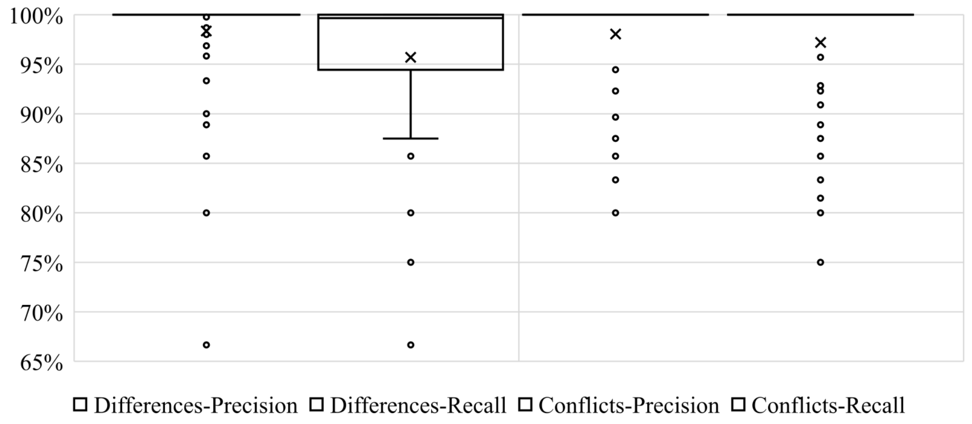


Figure 17 Precision and recall for detecting the semantic differences and conflicts

9.6.1. RQ1: DSMCompare effectiveness to find semantic differences.

According to the results, DSMCompare can find almost all semantic differences across all commits. It finds fine-grained and semantic differences of different types across different model sizes and various projects.

DSMCompare detects different kinds of *SDRule* patterns. It effectively detects semantic differences for rules focusing on simple attribute changes, relying on structural patterns, and requiring complex constraints to check. It also successfully detects semantic difference rules applied multiple times in the same and multiple versions of different projects. However, DSMCompare was unable to find a few refactoring patterns that require investigating structural content encoded as strings. It also incorrectly detected a few refactorings when the models were missing parts of the original source code.

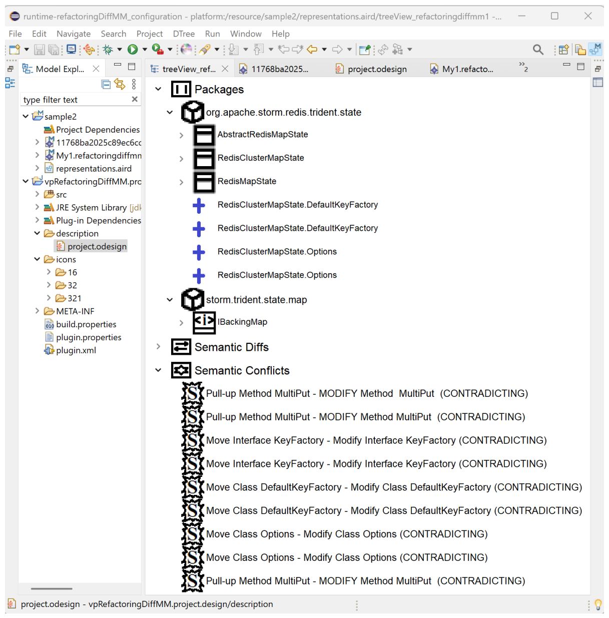
9.6.2. RQ2: DSMCompare effectiveness to find semantic conflicts.

DSMCompare can find almost all semantic conflicts across all conflicting commits. It finds different types and granularities of conflicts, including between semantic differences and fine-grained differences. Note that all missed conflicts correspond to missed differences.

DSMCompare also finds conflicts that occur in the same project and across different projects. For example, all projects have a conflict involving the *Extract Method* refactoring type. Some refactorings tend to be more conflicting even though they occur less often than other refactoring types. For example, as illustrated in Figure 16, *Rename Method* is responsible for 13% of all differences but contributes to 28% of all the conflicts.

9.6.3. Advantages of DSMCompare.

DSMCompare offers a more tailored display of differences that is specific to the relevant domain, and it is less verbose compared to *RefConfMiner* and EMFCompare. Additionally, it is important to mention that DSMCompare does not necessitate developers to create an ad-hoc metamodel. They can readily provide the metamodel of their DSL to utilize the tool. It visualizes the effect of syntactic differences on semantic changes. It also explicitly links the semantic difference instances to changed model elements. It reports the differences using the original DSL concrete


runtime-refactoringDiffMM_configuration - platform:/resource/sample2/representations.aird/treeView_refactoringdiffmm1 -...

```
sample2
  Project Dependencies
  11768ba2025c89ec6cc
  My1.refactoringdiffmn
  representations.aird
vpRefactoringDiffMM.prc
  src
  JRE System Library [jdk
  Plug-in Dependencies
  description
    project.odesign
  icons
    16
    32
    321
  META-INF
  build.properties
  plugin.properties
  plugin.xml
```

```
Packages
  org.apache.storm.redis.trident.state
    AbstractRedisMapState
    RedisClusterMapState
    RedisMapState
    RedisClusterMapState.DefaultKeyFactory
    RedisClusterMapState.DefaultKeyFactory
    RedisClusterMapState.Options
    RedisClusterMapState.Options
  storm.trident.state.map
    IBackingMap
Semantic Diffs
Semantic Conflicts
  Pull-up Method MultiPut - MODIFY Method  MultiPut  (CONTRADICTING)
  Pull-up Method MultiPut - MODIFY Method MultiPut  (CONTRADICTING)
  Move Interface KeyFactory - Modify Interface KeyFactory (CONTRADICTING)
  Move Interface KeyFactory - Modify Interface KeyFactory (CONTRADICTING)
  Move Class DefaultKeyFactory - Modify Class DefaultKeyFactory (CONTRADICTING)
  Move Class DefaultKeyFactory - Modify Class DefaultKeyFactory (CONTRADICTING)
  Move Class Options - Modify Class Options (CONTRADICTING)
  Move Class Options - Modify Class Options (CONTRADICTING)
  Pull-up Method MultiPut - MODIFY Method MultiPut  (CONTRADICTING)
```

project.odesign - vpRefactoringDiffMM.project.design/description

Figure 18 Visualization by Sirius, Tree-View presentation

syntax. Therefore, DSMCompare helps to understand and locate the exact problematic model elements conflicting with a semantic change. We claim that all these advantages help DSL users resolve conflicts more easily, save time, and increase the quality of the merged models.

As an indicator, the computation time of DSMCompare takes around one second for small MiniJava models (1–500 model elements), less than three seconds for medium models (500–1 000 model elements), and around nine seconds for large models (over 1 000 model elements). We ran the experiments on a Windows 10 machine with an i5-6300U processor clocked at 2.4 GHz, 8 GB of RAM on Eclipse version 4.24.0 with JDK 1.8 and the heap size set to 24 GB.

9.6.4. Multi-view Visualization.

As shown in Figure 15, the $Diff_{012}$ model is very cluttered visually when there are many model elements, fine-grained and semantic differences, and conflicts of different types, like in our dataset. To manage this problem, we use a layering mechanism (see Section 6) that DSL users can utilize to focus only on a specific part of the visualized differences and conflicts. However, sometimes, the number of differences and conflicts is just too large, and the adapted concrete syntax of the DSL needs to scale better in terms of readability. For example, if the DSL user wants to see the relations between models and the related differences and conflicts, the user will be presented with a disordered collection of graphical entities and associations among them. As a result, it obscures the semantics present in the Diff_{012} model.

To overcome this problem, we use a tree-view presentation to manage the complexity of visualizing when the number of differences and conflicts are very high. Figure 18 shows this alternative visualization of the same difference model presented in Figure 15. Here, we categorize the Diff_{012} model for MiniJava into three containers: the package container, the semantic difference container, and the conflict container. Under the package container, we collect all the packages and the hierarchy of the classes and sub-elements. The semantic difference container includes the list of all semantic differences and a description based on the type of the difference. The conflict container lists all the conflicts.

We integrate this view as an alternative representation using multiple views with a rich client platform[6] to visualize the Diff_{012} model. Using multiple views, the DSL user can search for semantic differences or conflicts on the tree-view and highlight the relevant model elements involved in the selected semantic difference or conflict. In this way, we enhance search time, search accuracy, perceived ease of use, and perceived usefulness (Adipat et al. 2011).

9.6.5. Comparing DSMCompare to other tools. Unlike DSMCompare, EMFCompare is not able to detect semantic differences by default. However, the tailor-made model comparison of EMFCompare provides custom filtering and domain-specific grouping features[7]. In addition, developers can add a new kind of difference, including a single fine-grained difference, to the comparison model of the EMFCompare. To have tailor-mode EMFCompare features, developers must manually add Java code as the plug-in extension points. However, it does not support creating semantic difference patterns referencing multiple elements and changes in the metamodel of the DSL. To the best of our knowledge, it is not possible to detect semantic conflicts in the current version of EMFCompare, even by adding custom code.

Both *RefactoringMiner* and *RefConfMiner* are text-based and can investigate refactoring changes in Java projects. *RefactoringMiner* reports refactorings, and *RefConfMiner* uses it to find refactorings involved in conflicts across the history of a Java project. They both provide high accuracy, especially when refactorings include text-based changes in method bodies, which is why we use them as the baseline for our assessment. In comparison, DSMCompare introduces mechanisms to find both semantic differences and conflicts. It also associates conflicts with relevant semantic differences and conflicts. The DSL engineer does not need to program how to find each refactoring type. Instead, she describes the patterns needed to specify the semantic differencing rule using the concrete syntax of the original DSL. DSMCompare also provides multiple domain-specific views to visualize the differences and conflicts between DSL users. Moreover, DSMCompare is agnostic of the programming/modeling language at hand. Like in our experiment, it can be used to work on models extracted from any programming language, not only Java projects.

9.7. Limitations and threats to validity

We outline some limitations of the experiment and our approach.

9.7.1. Threats to internal validity. Threats to the internal validity of this experiment are related to the assumptions we rely on.

We rely on the findings of EMFCompare to detect fine-grained differences and assume they are correct. As a result, any inaccurate preparatory information produced by EMFCompare influences the outcomes of both semantic differences and conflicts output by DSMCompare. However, EMFCompare is a trusted tool used by many model-based VCS, such as CDO, to benefit from its fine-grained comparison reports.

We manually checked all the outputs to ensure the semantic differences and conflicts we found by DSMCompare correspond to those found by *RefConfMiner*. However, this manual process can lead to human errors, which may threaten validity. Nevertheless, this process helped fix bugs in different parts of DSMCompare, which gives us confidence that the dataset is correct.

9.7.2. Threats to construct validity. Threats to the construct validity of this experiment are related to some of the tools we used in the experiment setup (see Figure 10). As we have explained earlier, there are three reasons to explain the presence of very few false positives and negatives in our results.

First, when transforming the Java source code into Ecore models, *MoDisco* creates a single string for each method body by concatenating the structure of the method. Therefore, refactorings affecting the internal structure of a method body, like *Inline Method*, cannot be captured in *SDRule* patterns for the MiniJava DSL. This prevents DSMCompare from correctly detecting these refactoring (false negatives). Switching to a different reverse engineering tool may help explicitly model the missing data.

Second, the same issue with MoDisco also resulted in some false positives in refactorings that are very similar, like *Extract Method* and *Extract and Move Method*. The strings used representing the extracted and added method bodies might occasionally be mistaken for one another. Every flow inside the extracted method body resembles a flow within another inserted method, though they differ in the finer details of their content. Because of this lack of information, DSMCompare mistakenly perceives the creation of a new method as other refactoring types.

[6] https://www.eclipsecon.org/europe2019/sessions/make-your-transition-cloud-tooling-now-thanks-hybrid-rcpweb-approach last accessed Jul 2022
[7] https://techconf.me/talks/35768 last accessed Jul 2022

The source code of the projects in our experiment consists of a large number of files and lines of code. To keep the dataset as concise as possible, we retrieved only the files involved in the merge conflicts. Thus, we relied on *MergeScenarioMiner* to download exclusively the Java files involved in the merge conflicts for all three versions of a conflicting three-way merge commit. However, this tool occasionally fails to download all the relevant files or downloads only parts of the linked files. This also prevents DSMCompare from correctly detecting some refactorings (false negatives). Fixing *MergeScenarioMiner* to obtain all the pertinent Java files or downloading the whole repository of all the involved versions in the conflicting merge commit may improve the results.

Nevertheless, as the overall results show, DSMCompare can find every other refactoring type and conflict related to classes, associations, methods, and attribute changes.

9.7.3. Threats to external validity.

Threats to the external validity of this experiment are related to the generalization of the results.

The results we present are specific to the dataset we created. Therefore, the results may be different for other datasets of refactoring commits or even on DSLs other than MiniJava. However, this dataset presents a wide diversity of cases with respect to *SDRules*, model sizes, semantic difference occurrences, and semantic/fine-grained conflicts. Moreover, the dataset originates from third-party programs. In Zadahmad et al. (2022), we evaluated DSMCompare on other DSLs as well.

Furthermore, there is a lack of openly accessible repositories of models with a commit history and, in particular, three-way difference conflicts. Our solution was to consider source code as models by reverse engineering repositories with these specificities.

9.7.4. Limitations.

Currently, DSMCompare generates editors for graphical DSLs only. Thus, it presents differences and conflicts in a graphical way only. Adaptations are needed to deal with textual concrete syntax. As we have seen in this experiment, graphical visualization of differences hits its limits when the models have a lot of elements.

The semantic differences that DSMCompare finds strongly depend on the *SDRules* provided by the DSL engineer. Thus, DSMCompare is only effective in providing semantic differences and conflicts if the rules are diverse enough to cover a variety of rule patterns, comprehensive enough to include all changes and conflicts at all granularities, and semantically relevant to the domain. Nevertheless, DSMCompare generates a domain-specific editor to enable DSL engineers to specify patterns for semantic differences. It also provides functionality to automatically create *SDRules* from two successive versions exhibiting a semantic change, which further helps DSL engineers.

We do not claim that the results of the experiment show that DSMCompare presents Java code refactoring better than existing tools (Dig et al. 2007). It is also not optimized to solve identify refactoring opportunities in programs. Nevertheless, in the given dataset, DSMCompare can detect refactoring instances and refactoring-induced conflicts on Ecore models that represent Java code.

10. Related work

10.1. Model differencing

Stephan & Cordy (2013) present a survey of several model comparison tools and methodologies. Some are specific to a modeling languages and others are metamodel-agnostic like DSMCompare. In that survey, EMFCompare uses a static identity-based comparison, is metamodel-agnostic and is applicable in real-world model versioning scenarios. This justifies our decision to rely on EMFCompare for the model matching phase and detection of fine-grained differences.

Schipper et al. (2009) extended EMFCompare to depict schematic differences in diagrams, which is comparable to our work. However, They only enable the visualization of atomic changes and do not support more coarse-grained changes or conflict patterns. Similarly, Cicchetti et al. (2010) generate model differences as model patches, but do not conflict analysis.

Several approaches have been proposed to semantically lift low-level changes, e.g., Kehrer et al. (2011, 2013) use Henshin for semantic lifting and critical pairs for dependency analysis. Langer et al. (2013) post-processes atomic operation into complex operations using EMFCompare. However, to work with EMFCompare extension points effectively, a DSL engineer should possess strong Java programming skills and a solid understanding of the Eclipse Platform and its extension mechanisms. Additionally, a good grasp of EMF core concepts, modeling principles, and model comparison and merge concepts is essential. Knowledge of XML and Ecore metamodeling, debugging techniques, design patterns, and testing methodologies are also valuable to ensure the successful implementation and customization of EMFCompare's comparison and merging capabilities. Our approach semantically lifts and conducts dependency analysis using multiCDA. We also visualize the differences and conflicts in concrete syntax.

Addazi et al. (2016) expanded the default matching process in EMFCompare to distinguish between linguistic and contextual notions, such as information-content based metrics. It provides a method for determining the semantic similarity between two given model elements. This somehow enables semantic reasoning over differences. Their solution managed to maintain fast time performance but did not deliver the best results in terms of precision and recall.

It should be noted that, we do not map models to a formally defined semantic domain in order to reason about the differences as done in Maoz et al. (2011). Rather, in the context of this paper, the term semantics pertains to editing semantics.

10.2. Conflict detection

Like DSMCompare, Brosch, Seidl, et al. (2012) create a separate $Diff_{012}$ model to represent different kinds and granularities of differences and conflicts. A difference is shown as a hierarchy divided into atomic changes (e.g., adding an element) and composite changes (e.g., refactoring). A conflict is shown as a hierarchy of overlapping conflicts (e.g., *DELETE/MODIFY* conflict) and constraint violations. However, they are specific to UML class diagrams.

Sharbaf et al. (2020) provide a conflict pattern language to specify conflicts in different modeling languages. In some sense, this is similar to the *SDRules* in DSMCompare. However, it only detects conflicting semantics and ignores non-conflicting semantics. Whereas, in our approach, we find semantic differences and investigate them for semantic conflicts. In their approach, the DSL engineer can express the changes in the metamodel elements between the different versions that lead to a conflict. The pattern language is built on top of OCL which restricts its application to UML-based languages only and forces the DSL engineer to be familiar with them. In DSMCompare, we extract complex change patterns from low-level model evolutions, much as semantic lifting techniques, similar to (García et al. 2013; Vermolen et al. 2012). However, their patterns are general and predefined, despite the fact that they resemble the rules in our method.

Sharbaf & Zamani (2020) use UML profiles to visualize changes and formalized constraints using OCL for UML models defined in Papyrus. They also highlight the conflicts using different colors. However, their approach is only appropriate for UML models, whereas DSMCompare supports any DSL. Furthermore, the static semantics of UML, which delegate model validation to the tools that process them, are currently insufficient to assure solid models (Berkenkötter 2008).

The tool PEACEMAKER is capable of loading XMI models with conflict sections, computing and displaying fine-grained conflicts at the model level, and offering the necessary resolution steps (de la Vega & Kolovos 2022). DSMCompare also load only conflicting parts of each three versions involved in the conflicting commits. Calculating potential conflicts a priori also improves the time performance. However, when using PEACEMAKER, DSL users must reason about differences and conflicts at the serialization level rather than using the concrete syntax of the DSL like in DSMCompare. Nevertheless, PEACEMAKER is able to resolve conflicts and merge the differences, which is not yet supported in DSMCompare.

Taentzer et al. (2014) use graph theory to formalize two syntax-based conflict concepts, including operation-and state-based conflicts in model versioning. Additionally, they use graph constraints to define multiplicity and ordered features. They detect conflicts using a set of conflicting operations such as *DELETE/MODIFY*. However, their approach disregards the effect of syntactic modifications on the semantics of the model as explained by Kautz & Rumpe (2018). Internally, DSMCompare also relies on the theory of graph transformation by executing Henshin rules. It also relies on MultiCDA (Lambers et al. 2019) to analyzing their dependencies and find potential conflicts between semantic and fine-grained differences.

To evaluate our approach, we utilized a sizable dataset of 288 Ecore models, which we have made publicly accessible. It is worth noting that there is a scarcity of existing repositories for domain-specific models, as noted in (Zadahmad et al. 2022). Brosch et al. (Brosch et al. 2010) proposed a web-based, collaborative conflict lexicon named Colex. However, we have found that the weblink associated with their proposal appears to be broken.

10.3. Versioning tools

Throughout the years, a number of model repositories with capabilities for version control have been introduced (Altmanninger et al. 2009). ChronoSphere (Haeusler et al. 2019) delivers an open-source EMF model repository. Transactions, queries, versioned persistence, and metamodel development are all part of the essential data management stack. The authors suggest using NoSQL databases for persistence for greater performance (Espinazo-Pagán et al. 2011). These repositories can be used in conjunction with DSMCompare to make it possible to visualize (semantic) differences using the graphical concrete syntax of the DSL.

Different levels of versioning and model differencing capabilities are available in commercial modeling programs. MagicDraw[8] provides controlled access to all artifacts, simple configuration management, and a mechanism to prevent version conflicts in this manner. Obeo Designer[9] and CDO can integrate with EMFCompare to provide a generic model-based versioning service. Smart Model Versioning[10], a version control tool included in MetaEdit+ (Kelly 2018), allows the comparison of models visually and textually. It is compatible with any significant VCS for storage, such as Git. Git and Subversion are integrated with JetBrains MPS, which also offers some tools for examining model differences textually. While these tools offer different ways to compare models triplets, they are typically not customizable to the DSL. DSMCompare provides domain-specific, customizable visualizations of the model differences, in a graphical way.

11. Conclusion

This paper introduces an approach for detecting fine-grained and semantic differences and conflicts based on a three-way comparison. Our solution is integrated into a new version of DSMCompare that previously only handled two-way domain-specific differences. It supports detecting and representing equivalent and contradicting conflicts between model versions. DSMCompare allows users to create semantic rules that automatically aggregate fine-grained differences and give domain-specific meaning to conflicts. Finally, we enhanced the concrete syntax to let DSL users visualize the three-way conflicts and differences more effectively. We evaluated our approach on multiple well-known open-source projects. The results demonstrate that DSMCompare is very effective at detecting semantic differences and conflicts with high accuracy. The large dataset of model versions involved in the commit history of several open-source projects and their labeled fine-grained and semantic differences and conflicts are also available for future research.

We plan to incorporate a conflict reconciliation mechanism in DSMCompare to automatically resolve the conflicts in the difference model and help the DSL user resolve them manually

[8] https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/ last accessed Jul 2022
[9] https://www.obeodesigner.com/ last accessed Jul 2022
[10] https://www.metacase.com/news/smart_model_versioning.html last accessed Jul 2022

when needed. This would lead to a final merged model free of conflicts that can be committed to a VCS repository. We also plan to integrate DSMCompare in domain-specific VCS to provide a fully-integrated system to DSL users. Another future line of research is to investigate how to represent domain-specific differences and conflicts for a textual DSL.

References

Addazi, L., Cicchetti, A., Di Rocco, J., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2016). Semantic-based model matching with emfcompare. In *Workshop on models and evolution* (Vol. 1706, pp. 40–49). CEUR-WS.org.

Adipat, B., Zhang, D., & Zhou, L. (2011). The effects of tree-view based presentation adaptation on mobile web browsing. *Mis Quarterly*, 99–121.

Altmanninger, K., Kappel, G., Kusel, A., Retschitzegger, W., Seidl, M., Schwinger, W., & Wimmer, M. (2008). Amor–towards adaptable model versioning. In *Workshop on model co-evolution and consistency management* (Vol. 8, pp. 4–50).

Altmanninger, K., Seidl, M., & Wimmer, M. (2009). A survey on model versioning approaches. *International Journal on Web Information Systems*, *5*(3), 271–304.

Berkenkötter, K. (2008). Reliable uml models and profiles. *Electronic Notes in Theoretical Computer Science*, *217*, 203–220.

Brosch, P., Kappel, G., Seidl, M., Wieland, K., Wimmer, M., Kargl, H., & Langer, P. (2010). Adaptable Model Versioning in Action. In *Modellierung 2010* (Vol. 161, pp. 221–236).

Brosch, P., Langer, P., Seidl, M., Wieland, K., & Wimmer, M. (2010). Colex: a web-based collaborative conflict lexicon. In *Proceedings of the 1st international workshop on model comparison in practice* (pp. 42–49).

Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., & Kappel, G. (2012). The past, present, and future of model versioning. In *Emerging technologies for the evolution and maintenance of software models* (pp. 410–443). IGI Global.

Brosch, P., Seidl, M., Wimmer, M., & Kappel, G. (2012). Conflict visualization for evolving UML models. *Journal of Object Technology*, *11*(3), 2:1–30.

Brun, C., & Pierantonio, A. (2008). Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, *9*(2), 29–34. (https://www.eclipse.org/emf/compare/)

Bruneliere, H., Cabot, J., Dupé, G., & Madiot, F. (2014). Modisco: A model driven reverse engineering framework. *Information and Software Technology*, *56*(8), 1012–1032.

Cicchetti, A., Di Ruscio, D., & Pierantonio, A. (2010). Model patches in model-driven engineering. In *Models in software engineering: Workshops and symposia at models 2009, denver, co, usa, october 4-9, 2009, reports and revised selected papers 12* (pp. 190–204).

David, I., Aslam, K., Faridmoayer, S., Malavolta, I., Syriani, E., & Lago, P. (2021). Collaborative Model-Driven Software Engineering: A Systematic Update. In *Model driven engineering languages and systems* (pp. 273–284). ACM.

de la Vega, A., & Kolovos, D. (2022). An efficient line-based approach for resolving merge conflicts in XMI-based models. *Software and Systems Modeling*, 1–27.

Dig, D., Manzoor, K., Johnson, R., & Nguyen, T. N. (2007). Refactoring-aware configuration management for object-oriented programs. In *International conference on software engineering* (pp. 427–436).

Eclipse EGit. (2023). https://www.eclipse.org/egit/. ((last accessed in March 2023))

EMF Compare. (last accessed January 2021). https://www.eclipse.org/emf/compare/.

EMF DiffMerge. (2023). https://wiki.eclipse.org/EMF_DiffMerge. ((last accessed in March 2023))

EMFCompare. (2023). https://techconf.me/talks/35768. ((last accessed in March 2023))

Espinazo-Pagán, J., Cuadrado, J. S., & Molina, J. G. (2011). Morsa: A scalable approach for persisting and accessing large models. In *Model-driven language engineering and systems* (Vol. 6981, pp. 77–92). Springer.

García, J., Diaz, O., & Azanza, M. (2013). Model transformation co-evolution: A semi-automatic approach. In *Software language engineering* (Vol. 7745, pp. 144–163). Springer.

Haeusler, M., Trojer, T., Kessler, J., Farwick, M., Nowakowski, E., & Breu, R. (2019). Chronosphere: a graph-based EMF model repository for IT landscape models. *Software and Systems Modeling*, *18*(6), 3487–3526.

Jackson, D., & Ladd, D. A. (1994). Semantic diff: a tool for summarizing the effects of modifications. In *International conference on software maintenance* (pp. 243–252). IEEE.

Kautz, O., & Rumpe, B. (2018). On computing instructions to repair failed model refinements. In *Model driven engineering languages and systems* (pp. 289–299).

Kehrer, T., Kelter, U., & Taentzer, G. (2011). A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Automated software engineering* (pp. 163–172). IEEE Computer Society.

Kehrer, T., Kelter, U., & Taentzer, G. (2013). Consistency-preserving edit scripts in model versioning. In *2013 28th ieee/acm international conference on automated software engineering (ase)* (pp. 191–201).

Kelly, S. (2018). Collaborative modelling with version control. In *Software technologies: Applications and foundations* (Vol. 10748, pp. 20–29). Springer.

Kelly, S., & Tolvanen, J.-P. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons.

Koegel, M., & Helming, J. (2010). EMFStore: a model repository for EMF models. In *International conference on software engineering* (Vol. 2, pp. 307–308). ACM.

Kolovos, D. S. (2009). Establishing correspondences between models with the epsilon comparison language. In *Model driven architecture-foundations and applications: 5th european conference, ecmda-fa 2009, enschede, the netherlands, june 23-26, 2009. proceedings 5* (pp. 146–157).

Kramler, G., Kappel, G., Reiter, T., Kapsammer, E., Retschitzegger, W., & Schwinger, W. (2006). Towards a semantic infrastructure supporting model-based tool integration. In *Workshop on global integrated model management* (p. 43—46).

ACM.

Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., & Wimmer, M. (2009). Explicit transformation modeling. In *MODELS 2009 workshops* (Vol. 6002, pp. 240–255). Springer.

Lambers, L., Born, K., Kosiol, J., Strüber, D., & Taentzer, G. (2019). Granularity of conflicts and dependencies in graph transformation systems: a two-dimensional approach. *Journal of logical and algebraic methods in programming*, *103*, 105–129.

Lambers, L., Ehrig, H., & Orejas, F. (2008). Efficient conflict detection in graph transformation systems by essential critical pairs. *Electronic Notes in Theoretical Computer Science*, *211*, 17–26.

Lambers, L., Strüber, D., Taentzer, G., Born, K., & Huebert, J. (2018). Multi-granular conflict and dependency analysis in software engineering based on graph transformation. In *International conference on software engineering* (pp. 716–727).

Langer, P., Wimmer, M., Brosch, P., Herrmannsdörfer, M., Seidl, M., Wieland, K., & Kappel, G. (2013). A posteriori operation detection in evolving software models. *Journal of Systems and Software*, *86*(2), 551–566.

Leßenich, O., Siegmund, J., Apel, S., Kästner, C., & Hunsen, C. (2018). Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering*, *25*(2), 279–313.

Lin, Y., Gray, J., & Jouault, F. (2007). Dsmdiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, *16*(4), 349–361.

Mahmoudi, M., Nadi, S., & Tsantalis, N. (2019). Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *Software analysis, evolution and reengineering* (pp. 151–162). IEEE.

Maoz, S., Ringert, J. O., & Rumpe, B. (2011). Addiff: semantic differencing for activity diagrams. In *Proceedings of the 19th acm sigsoft symposium and the 13th european conference on foundations of software engineering* (pp. 179–189).

Munaiah, N., Kroh, S., Cabrey, C., & Nagappan, M. (2017). Curating github for engineered software projects. *Empirical Software Engineering*, *22*(6), 3219–3253.

Owhadi-Kareshk, M., Nadi, S., & Rubin, J. (2019). Predicting merge conflicts in collaborative software development. In *Symposium on empirical software engineering and measurement* (pp. 1–11). IEEE.

Pinto, G., Steinmacher, I., Dias, L. F., & Gerosa, M. (2018). On the challenges of open-sourcing proprietary software projects. *Empirical Software Engineering*, *23*(6), 3221–3247.

Rivera, J. E., & Vallecillo, A. (2008). Representing and operating with model differences. In *Objects, components, models and patterns: 46th international conference, tools europe 2008, zurich, switzerland, june 30-july 4, 2008. proceedings 46* (pp. 141–160).

Rubin, J., & Chechik, M. (2013). N-way model merging. In *proceedings of the 2013 9th joint meeting on foundations of software engineering* (pp. 301–311).

Schipper, A., Fuhrmann, H., & von Hanxleden, R. (2009). Visual comparison of graphical models. In *International conference on engineering of complex computer systems* (pp. 335–340). IEEE.

Schultheiß, A., Bittner, P. M., Grunske, L., Thüm, T., & Kehrer, T. (2021). Scalable n-way model matching using multi-dimensional search trees. In *2021 acm/ieee 24th international conference on model driven engineering languages and systems (models)* (pp. 1–12).

Sharbaf, M., & Zamani, B. (2020). Configurable three-way model merging. *Software: Practice and Experience*, *50*(8), 1565–1599.

Sharbaf, M., Zamani, B., & Sunyé, G. (2020). A formalism for specifying model merging conflicts. In *System analysis and modelling conference* (pp. 1–10).

Shen, B., Zhang, W., Yu, A., Shi, Y., Zhao, H., & Jin, Z. (2021). Somanyconflicts: Resolve many merge conflicts interactively and systematically. In *Automated software engineering* (pp. 1291–1295). IEEE.

Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z., & Wang, Q. (2019). IntelliMerge: a refactoring-aware software merging technique. *Programming Languages*, *3*, 1–28.

Sirius. (2023). https://www.eclipsecon.org/europe2019/sessions/make-your-transition-cloud-tooling-now-thanks-hybrid-rcpweb-approach. ((last accessed in March 2023))

Stephan, M., & Cordy, J. R. (2013). A survey of model comparison approaches and applications. In *Modelsward* (pp. 265–277). SciTePress.

Strüber, D., Born, K., Gill, K. D., Groner, R., Kehrer, T., Ohrndorf, M., & Tichy, M. (2017). Henshin: A usability-focused framework for emf model transformation development. In *International conference on graph transformation* (pp. 196–208). Springer.

Taentzer, G., Ermel, C., Langer, P., & Wimmer, M. (2014). A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling*, *13*(1), 239–272.

Toyoshima, I., Yamaguchi, S., & Zhang, J. (2015). A refactoring algorithm of workflows based on petri nets. In *International congress on advanced applied informatics* (pp. 79–84). IEEE. doi: 10.1109/IIAI-AAI.2015.273

Tsantalis, N., Ketkar, A., & Dig, D. (2020). Refactoringminer 2.0. *Transactions on Software Engineering*, *48*(3), 930-950.

van der Aalst, W. M. P. (1998). The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, *8*(1), 21–66. doi: 10.1142/S0218126698000043

Vermolen, S. D., Wachsmuth, G., & Visser, E. (2012). Reconstructing complex metamodel evolution. In *Software language engineering* (Vol. 6940, pp. 201–221). Springer.

Viyović, V., Maksimović, M., & Perisić, B. (2014). Sirius: A rapid development of dsm graphical editor. In *Intelligent engineering systems* (pp. 233–238). IEEE.

Zadahmad, M., Syriani, E., Alam, O., Guerra, E., & de Lara, J. (2019). Domain-specific model differencing in visual concrete syntax. In *Software language engineering* (pp. 100–112). ACM.

Zadahmad, M., Syriani, E., Alam, O., Guerra, E., & de Lara, J. (2022). DSMCompare: Domain-specific model differencing for graphical domain-specific languages. *Software & Systems*

Modeling, *21*, 2067–2096.

About the authors

Manouchehr Zadahmad Jafarlou is a Ph.D. student in the department of computer science and operations research at Université de Montréal. His main research is on model differncing and merging. You can contact him at Manouchehr.zadahmad.jafarlou@umontreal.ca or visit https://www.linkedin.com/in/manouchehr-zadahmad/.

Eugene Syriani is an associate professor in the department of computer science and operations research at Université de Montréal. His main research interests fall in software design based on the model-driven engineering approach, the engineering of domain-specific languages, model transformation and code generation, simulation-based design, collaborative modeling, and user experience. You can contact him at syriani@iro.umontreal.ca or visit www.iro.umontreal.ca/~syriani.

Omar Alam is an associate professor in the department of computer science at Trent University. His area of research falls in model-driven engineering, model reuse, collaborative modeling, software modularity and computing education. You can contact him at omaralam@trentu.ca or visit www.omaralam.org.