Université de Montréal

Domain-specific differencing and merging of models

par

Manouchehr Zadahmad Jafarlou

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en Informatique - Génie Logiciel

novembre 2023

Université de Montréal

Faculté des études supérieures et postdoctorales

Cette thèse intitulée

Domain-specific differencing and merging of models

présentée par

Manouchehr Zadahmad Jafarlou

a été évaluée par un jury composé des personnes suivantes :

Michalis Famelis
(président-rapporteur)

Eugene Syriani
(directeur de recherche)

Omar Alam
(co-directeur)

Abdelhakim Hafid
(membre du jury)

Dimitris Kolovos
(examinateur externe)

—
(représentant du doyen de la ESP)

Thèse acceptée le :
31 August 2023

Sommaire

En génie logiciel collaboratif, les systèmes de contrôle de version (SCV) jouent un rôle crucial dans la gestion des changements de code, la promotion de la collaboration et la garantie de l'intégrité des projets partagés. Cette importance s'étend à l'ingénierie dirigée par les modèles (IDM), où les experts du domaine conçoivent des modèles spécifiques au domaine (MSD). Dans ce contexte, la collaboration avec les SCV permet de coordonner les changements de modèles et de préserver l'intégrité des MSD. Cependant, les solutions existantes se concentrent principalement sur des approches génériques, considérant les modèles comme du texte générique. Ces SCV rapportent les différences entre les versions des modèles d'une manière abstraite et non intuitive pour les experts du domaine. Cela pose également des défis lors de la résolution des conflits et de la fusion des modèles, ce qui ajoute de la complexité au flux de travail des experts du domaine.

L'objectif de cette thèse est de fournir des SCV spécifiques à un domaine donné en se concentrant sur les deux principaux composants des SCV, à savoir la différenciation et la fusion. Nous présentons DSMCompare, un outil de comparaison de modèles spécifique au domaine, intégré avec des capacités de détection, de résolution et de fusion de conflits de triplets de versions. DSMCompare fournit des représentations concises des différences et conflits à différents niveaux de granularité, tout en utilisant la syntaxe graphique des MSD originaux. Dans nos évaluations, DSMCompare a démontré des améliorations notables par rapport aux solutions génériques de différenciation et de fusion, notamment une réduction de la verbosité des différences rapportée, des différences exprimée en utilisant la sémantique du domaine, une détection précise des différences sémantiques et des conflits entre différentes versions d'un modèle, une résolution correcte des conflits, une diminution des interactions manuelles requises et une amélioration globale de l'efficacité pour les experts du domaine.

Mots clefs : Ingénierie logicielle, Ingénierie dirigée par les modèles, Modélisation spécifique au domaine, Systèmes de gestion de version, Différenciation sémantique, Conflits sémantiques, Détection de conflits, Résolution de conflits, Fusion spécifique au domaine

Summary

In the context of collaborative software engineering, version control systems (VCS) play a crucial role in managing code changes, promoting collaboration, and ensuring the integrity of shared projects. This significance extends to model-driven engineering (MDE), where domain experts design domain-specific models (DSM). In this context, collaborating with VCS aids in coordinating model changes and preserving the integrity of DSMs. However, existing solutions primarily focus on generic approaches, considering models as generic text. VCS report the differences between model versions in an abstract and unintuitive way for domain experts. This also poses challenges when resolving conflicts and merging models, adding complexity to the workflow of domain experts.

The goal of this thesis is to provide domain-specific VCS for domain experts, focusing on the two main components of VCS, namely differencing and merging. We introduce DSMCompare, a domain-specific model comparison tool integrated with three-way conflict detection, resolution, and merging capabilities. DSMCompare provides concise representations of differences and conflicts at different levels of granularity, while using the graphical syntax of the original DSMs. In our evaluations, DSMCompare demonstrated significant improvements over generic differencing and merging solutions, including a reduction in reported difference verbosity, differences expressed using the semantics of the domain, accurate detection of semantic differences and conflicts between different versions of a model, correct conflict resolution, a reduction in manual interactions needed, and an overall improvement in efficiency for domain experts.

Keywords: Software engineering, Model-driven engineering, Domain-specific modeling, Version control systems, Semantic differencing, Semantic conflicts, Conflict detection, Conflict resolution, Domain-specific merging

Contents

List of Tables

List of Figures

Acknowledgments

I would like to express my sincere gratitude to Professor Doctor Eugene Syriani and my supervisor, Omar Alam, for their unwavering support, invaluable guidance, and scholarly insights throughout the course of my thesis. Their mentorship has been instrumental in shaping this work.

I am deeply thankful to my wife, Parisa, and my son, Atila, for their unwavering love, understanding, and patience during this demanding journey. Your encouragement sustained me through the long hours of research and writing. This work is dedicated to you with heartfelt appreciation.

To my parents and siblings, your constant encouragement and belief in my abilities have been a source of strength and inspiration.

I would also like to extend my gratitude to all my academic colleagues and friends who have shared their knowledge, ideas, and encouragement. Your contributions have enriched my research and personal growth.

I want to express my special thanks to Esther Guerra and Juan de Lara for collaborating with me on my first paper, their expertise and collaboration have been invaluable.

Lastly, I want to express my thanks to the Geodes laboratory for providing me with the resources and environment necessary to conduct my research effectively.

This work would not have been possible without the support and encouragement of all those mentioned above, and for that, I am truly grateful.

Chapter 1

Introduction

1. Context

Collaborative Software Engineering (Franzago et al, 2017) is an approach to software development where team members work together to design, develop, test, and maintain software applications. It involves coordinating efforts, sharing resources, and contributing to the development process in a collaborative manner (David et al, 2021). In Collaborative Software Engineering, various artifacts related to software development can undergo changes. Changes can range from simple edits to more complex modifications that require coordination among team members. Collaborative Software Engineering aims to manage these changes effectively, ensuring that team members can work together seamlessly. Moreover, Collaborative Software Engineering needs to ensure the artifacts remain consistent and aligned with the intended goals.

Software modeling is an integral part of collaborative software engineering, helping teams in their understanding and decision-making processes. It helps to create abstract representations of a software system (Rumbaugh et al, 1991). These representations, called models, capture various aspects of the software's structure, behavior, and functionality. Software models help developers, designers, and stakeholders to visualize, analyze, and communicate different aspects of the software before the actual implementation begins. These models can range from high-level architectural diagrams to detailed representations of specific components or processes.

Model-Driven Engineering (MDE) (Kent, 2002), is a methodology where models and software modeling techniques play a central role throughout the entire lifecycle of a project.

MDE also involves using tools to automatically generate code and other parts of the software from these models, making the development process more efficient and increasing quality.

Domain-specific modeling (DSM) is a key concept within MDE, emphasizing the practice of creating specialized modeling languages and tools tailored to specific problem domains (Kelly and Tolvanen, 2008). Unlike general-purpose modeling languages like UML, which are designed to cover a wide range of domains, domain-specific modeling languages (DSMLs) are customized to address the unique requirements and concepts of a particular domain. DSMs allow domain experts, who may not have extensive programming or software engineering backgrounds, to create and work with models that directly represent their domain knowledge. As DSMLs enable models to be expressed in a more intuitive and natural way, this enhances communication between domain experts and software developers.

Since models are accessed and used collaboratively, they undergo changes and therefore need to be versioned (Brosch et al, 2012a; Paige et al, 2016). To address this requirement, Version Control Systems (VCS) play a pivotal role (Franzago et al, 2017).VCS provides a systematic way to manage changes in software projects, ensuring that collaboration remains coherent and effective (Mens, 2002). In the context of Collaborative Software Engineering, VCSs serve as tools to track changes in all kinds of artifacts from models to source code.

Well-known VCSs such as Git (Git, last accessed 2023) or SVN (SVN, last accessed 2023) report the differences in a line-by-line or block-by-block manner. The report shows additions, deletions, and modifications made to the files over time. On the other hand, model versioning refers to the practice of managing different versions of models throughout their lifecycle. Several tools are available for model versioning, each offering different features and capabilities to help manage and track changes to models. Some of the notable tools for generic model versioning include *EMFStore* (Koegel and Helming, 2010a), *EMFCompare* (EMF Compare, accessed August 2023), and *CDO* (Connected Data Objects) (CDO Model repository, accessed August 2023).

In addition to keeping a record of changes, VCS provides different functionalities such as Differencing, Branching, and Merging. `Differencing` refers to the process of identifying and highlighting the differences between two versions of a file (code, model, or any file type) or a set of files. In two-way differencing, the comparison is performed between two distinct versions, while three-way differencing involves evaluating changes in three versions, typically a common ancestor and two divergent branches, facilitating more sophisticated conflict resolution in collaborative development scenarios.

The `difference` (Diff) refers to the changes between two versions of a file or code. Differencing can produce one or multiple differences. Therefore, Differencing allows developers to review, understand, and manage changes efficiently. Branching, on the other hand, creates a snapshot of the project including models at a specific point in time. It allows teams to work on different features or aspects in isolation. Changes made within a branch do not impact other branches.Branches can later be merged to incorporate the changes seamlessly.

2. Problem Statement

Although certain VCSs have been specifically designed for models (Altmanninger et al, 2008b; Koegel and Helming, 2010b), most practitioners opt for text-based VCSs like Git and SVN. However, these VCS are not ideal for effectively visualizing the changes in a model's versions in a way that is easily comprehensible (Zadahmad et al, 2019), as they do not grasp the syntax and semantics of the DSL. Generic model-based differencing tools, such as EMFCompare (Brun and Pierantonio, 2008), provide results that highlight differences in classes, attributes, and associations. Nevertheless, these results are presented in the abstract syntax of the DSL, which might not be familiar to DSL users. Additionally, presenting the fine-grained differences of a large model can be overwhelming for DSL users, as they cannot perceive the semantics of the changes (Sharbaf et al, 2022b). Thus, there exists a need for model-based differencing tools capable of presenting difference results in a more user-friendly manner tailored to DSL users.

In the context of domain-specific modeling, tools, languages, and methodologies are tailored to suit the specific characteristics and requirements of a particular domain. The goal is to create modeling languages and tools that closely align with the concepts, terminology, and processes of the target domain. It makes it easier for domain experts to create models and collaborate effectively.

However, tailoring DSM to a specific application domain presents unique challenges when using text-based VCSs (VCS) like SVN. DSM involves customizing modeling languages and tools to align with the domain's specific needs. However, SVN, as a text-based VCS, does not naturally understand the semantics and structure of DSM artifacts (Zadahmad et al, 2022). Generic model-based approaches also struggle with grasping the semantics of DSM artifacts (Langer et al, 2013b). This limitation hinders its ability to effectively detect and visualize domain-specific differences (Cicchetti et al, 2007). As a result, traditional methods can produce complex reports when there are small changes (Altmanninger et al, 2008a), making it difficult to notice important differences in meaning within the specific area. On the other hand, tools designed for versioning of domain-specific models do not provide a comprehensive solution for differencing and merging domain-specific models (Sharbaf et al, 2022b).

Models' evolution naturally leads to conflicts (Hachemi and Ahmed-Nacer, 2020). In DSM, these conflicts often exceed syntax to involve complicated semantic differences (Sharbaf et al, 2020). In this case, generic model-based or test-based VCS can struggle with semantic conflicts because they often focus on tracking syntactic changes, such as code refactorings in the class diagram design, rather than understanding the meaning or semantics behind those changes.

Semantic differences in a domain refer to changes that affect the underlying logic, structure, or meaning of code. These differences could involve various refactorings, such as method extractions, renamings, or changes in class hierarchies. The role of semantic differences is crucial because they can impact how the software functions, even if the syntactic changes are relatively small. For example, performing a method extraction might seem like a minor syntactic change, but it could significantly alter the organization and behavior of the code. This is particularly evident when the extracted method interacts differently with other parts of the code or when it leads to changes in the control flow.

VCS struggles with these conflicts because they often lack the ability to understand the semantic implications of such refactorings. They treat them as syntactic changes and may not provide effective tools for detecting, visualizing, or resolving these conflicts in a way that aligns with the domain-specific semantics.

Semantic conflicts, in this context, impact the core logic, structure, or meaning of the domain model. This can lead to incorrect conflict resolutions or missed conflicts, which can impact the reliability and quality of the software (Brosch et al, 2012f). To address these issues effectively, conflict resolution and merging tools should be equipped to recognize and manage these semantic differences, ensuring meaningful resolutions and maintaining the overall integrity of the codebase (Hachemi and Ahmed-Nacer, 2020). The proposed approach also needs to leverage effective visualization to manage the requirements of semantic differences, semantic conflicts, resolution, and merging complexities in the DSM (Sharbaf et al, 2022a; Langer et al, 2013b). We formulate three pivotal Research Questions (RQs) that guide our investigation:

RQ1: How can semantic differencing and visualization be enhanced to extract meaningful differences in a domain-specific manner, reduce verbosity, and provide DSL users with an intuitive understanding of changes?

RQ2: How can the effective detection and visualization of semantic differences and semantic conflicts in three-way domain-specific differencing be achieved, enabling the identification and clear visualization of various types of conflicts for efficient resolution and decision-making?

RQ3: How can conflict resolution in domain-specific contexts be empowered to assist DSL users in navigating, resolving, and reversing conflict resolutions effectively, considering the unique aspects of domain-specific conflicts to enhance collaboration and project advancement?

3. Contributions

In this section, we provide a brief overview of our primary contributions. The comprehensive details regarding the challenges and contributions can be found in Chapter 3. This thesis follows a thesis by article format whereby the core contributions are articles published or submitted to journals.

List of publications. The following scientific articles are integral to this thesis.

(1) Zadahmad M, Syriani E, Alam O, Guerra E, de Lara J. DSMCompare: domain-specific model differencing for graphical domain-specific languages. Software and Systems Modeling. 2022 Oct 1:1-30. Published. (Chapter 4)

(2) Zadahmad M, Syriani E, Alam O. From two-way to three-way: domain-specific model differencing and conflict detection. Journal of Object Technology. 2023, 1:1-29. Published. (Chapter 5)

(3) Zadahmad M, Syriani E, Alam O. Domain-specific conflict resolution and model merge. Journal of Systems and Software. To be submitted. (Chapter 6)

I have also published the following articles during my thesis.

(1) Zadahmad M, Syriani E, Alam O, Guerra E, de Lara J. Domain-specific model differencing in visual concrete syntax. InProceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering. 2019 Oct 20 (pp. 100-112). Published.

(2) Jafarlou MZ. Domain-specific model differencing for graphical domain-specific languages. InProceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. 2022 Oct 23 (pp. 205-208). Published.

We provide a brief overview of the main contributions of this thesis.

3.1. Enhancing Semantic Differencing and Visualization

In the initial phase, the goal is to advance techniques for extracting semantic differences and presenting them in a domain-specific manner. The proposed approach, DSMCompare, considers both abstract and concrete syntax of a DSL, supporting the definition of domain-specific semantics for specific difference patterns. Contributions include representation of model differences within a single DSL, a domain-specific semantic differencing rule editor, automated representation of model differences using graphical concrete syntax, and prototype tool support.

3.2. Effective Detection and Visualization of Semantic Conflicts

The focus shifts to three-way domain-specific differencing and conflict detection, aiming to identify and visualize various semantic conflicts comprehensibly. DSMCompare transitions from a two-way to a three-way model comparison, providing a domain-specific conflict detection mechanism, semantic differencing rule editor, visualization support with graphical concrete syntax, and tool implementation with extensive evaluation.

3.3. Empowering Conflict Resolution in Domain-Specific Contexts

In the final step, the objective is to empower DSL users with effective conflict resolution tools. The proposed domain-specific approach for three-way model merging includes a conflict resolution mechanism, algorithms for automated conflict resolution, user-friendly conflict resolution interfaces, implementation in DSMCompare, effectiveness comparison with other merging techniques, and an applicability study in practical settings.

4. Thesis Structure

The structure of this document encompasses five chapters and a concluding section:

Chapter II: The state-of-the-art (SOTA) section is outlined here.

Chapter III: This chapter outlines the challenges and the contributed approaches.

Chapter IV: In this chapter, the first article is presented. It introduces DSM-Compare, our dedicated tool for domain-specific model differencing within graphical DSMLs.

Chapter V: The focus of this chapter is the second article. It elaborates on the transition from two-way to three-way domain-specific tooling for differencing and discusses domain-specific model differencing and conflict detection within a three-way context.

Chapter VI: This chapter centers around the third article. It encompasses domain-specific conflict resolution and model merging.

Chapter VII: The concluding **conclusion** chapter offers a comprehensive overview of our work.

Chapter 2

Background and state of the art

In this chapter, we explore the state of the art regarding domain-specific model differencing and merging. We explore various aspects of MDE, DSML, VCS, and the existing literature on model versioning, comparison, conflict detection, resolution, and model merging. Our investigation highlights the challenges and shortcomings in current approaches, setting the stage for proposing novel techniques to enhance semantic model differencing, conflict management, and model merging tailored to specific domain needs.

1. MDE

These models serve as blueprints for generating executable code, documentation, and other artifacts, effectively bridging the gap between system design and implementation. In MDE, modeling languages play a crucial role as they provide the means to define and express these models effectively. Here are some of the modeling languages commonly used in MDE:

- *Unified Modeling Language (UML).* UML (UML, 2023) is one of the most well-known modeling languages in MDE. It provides a standardized way to represent various aspects of software systems, including classes, objects, relationships, and behavior. UML diagrams, such as class diagrams, sequence diagrams, and state diagrams, are commonly used in software modeling.
- *DSMLs.* DSMLs (Kelly and Tolvanen, 2008) are specialized modeling languages tailored to specific application domains. They are designed to capture domain-specific concepts, notations, and semantics. DSMLs allow developers to create models that closely align with the requirements and characteristics of a particular domain.

1.1. Abstraction

Abstraction is a fundamental concept in MDE. Software systems can be incredibly intricate, involving numerous components, interactions, and details (Brambilla et al, 2012). Abstraction allows developers to create models that capture the essential features and behaviors of a system while omitting unnecessary details. This simplification makes it easier to understand, analyze, and communicate about the system.

Abstraction raises the level of representation from the low-level details of code to higher-level models. Instead of working directly with code, developers use models to represent system architecture, design patterns, business processes, and other aspects. These models are more intuitive for stakeholders who may not have a deep technical understanding.

Moreover, abstraction allows different aspects of a system, such as its structure, behavior, and data, to be modeled independently. This separation enhances modularity and maintainability, as changes in one concern do not necessarily impact others.

1.2. Automation

Automation is a cornerstone of MDE, driving efficiency and precision in software development (Frankel, 2002). In MDE, automation refers to the automatic generation of artifacts, such as code, from high-level models. This process streamlines development by reducing manual intervention, minimizing errors, and ensuring consistency across different stages of the software lifecycle.

Code generation is a specific aspect of automation in MDE where software code is automatically produced from higher-level models. This can include generating code for various programming languages, platforms, and technologies.

1.3. Transformation

Transformation enables the automatic conversion of models from one representation to another. In MDE, models serve as the central artifacts that capture system specifications (Völter and Kelly, 2013). However, to bring these specifications to life, they must be transformed into executable code or other desired formats. Transformation processes ensure that the semantics of the high-level models are preserved while generating lower-level artifacts.

Model to model transformation refers to the process of converting one model into another model. Model to model transformation allows to express how information in one model relates to information in another, enabling the manipulation and synchronization of models. For example Henshin is an open-source model transformation language and toolset used in the field of MDE.

Henshin (Strüber et al, 2017) provides a DSL for specifying model transformations. It allows to define rules that describe how one model can be transformed into another model. These rules are typically expressed in a high-level, declarative manner. Henshin's conflict and dependency analysis (MultiCDA) feature and critical pair analysis (CPA) feature enables the detection of potential conflicts and dependencies of a set of rules

Atlas Transformation Language (ATL) (Eclipse Foundation, 2023a) is another model transformation language that allows developers to define and execute transformations between different models. It helps automate tasks like code generation, data mapping, and model synchronization.

But Henshin and ATL are two different kind of model to model languages. ATL is a declarative language. Developers specify what the transformation should achieve, and the ATL transformation engine determines how to achieve it. In contrast, Henshin is an pattern-based language. It allows developers to specify not only what should be transformed but also how the transformation should be performed in terms of actions and operations. ATL primarily uses rule-based transformations. Developers define transformation rules that specify how elements in the source model are mapped to elements in the target model. In contrast, Henshin uses a graph-based approach. Transformations are defined in terms of graph patterns that match elements in the source model, and these patterns are then replaced or modified to produce the target model.

Model to Code transformation is a specific type of model transformation that focuses on generating executable code from a model. In this process, a high-level model is transformed into code in a programming language. This can significantly speed up the development process and help maintain consistency between the model and the code.

For example, Epsilon Generation Language (Eclipse Foundation, 2023c) is a model to code language (model-to-text transformation in a broader view) that can be used to transform models into various types of textual artefact, including code (e.g. Java), reports (e.g. in

HTML/LaTeX), images (e.g. using Graphviz), formal specifications, or even entire applications comprising code in multiple languages (e.g. HTML, Javascript and CSS). Xtend (xtend, last accessed 2023) as a widely adopted model-to-text transformation language, frequently used in conjunction with Xtext (Xtext, last accessed 2023). While Xtext is utilized to define textual DSL, Xtend facilitates code generation from models defined using Xtext-defined DSLs, allowing developers to define templates and expressions that generate textual output based on input models. Another noteworthy tool in this domain is ATL (Eclipse Foundation, 2023a) that supports model-to-model transformations. It is primarily a rule-based transformation language. Transformations are defined by specifying declarative transformation rules, which describe how elements in the source model are mapped to elements in the target model.

Graph transformation is a technique used in MDE to manipulate models represented as graphs (Ehrig et al, 2006). In this context, a graph represents the elements of a model and their relationships. Graph transformation rules define how the graph can be modified. Graph transformation is versatile and can be used for various model manipulations, including refinement, refactoring, and analysis. For example, suppose you have a model representing a state machine. Using graph transformation rules, it is possible to define how transitions between states can be modified, added, or removed. This allows you to refactor and optimize the state machine model.

1.4. Frameworks

Frameworks play a crucial role in MDE, providing essential infrastructures and tools for modeling, transformation, and code generation. One of the prominent frameworks in the MDE ecosystem is the Eclipse Modeling Framework (EMF) (Eclipse Foundation, 2023b). EMF offers a comprehensive platform for developing and deploying model-based software applications. It provides a structured approach to defining data models and generating code, allowing developers to work at higher levels of abstraction. EMF is a part of the Eclipse Modeling Project. Main Components of EMF include:

- *Ecore*: At the heart of EMF is Ecore (EMF Core), which is a domain modeling technology. Ecore provides the foundation for defining and working with structured data models. Its main components include EClass (used to define the types of objects in a model), EAttribute (represent the properties or attributes of EClasses), EReference (define relationships between EClasses, similar to associations in UML), and EPackage (organize EClasses, EAttributes, and EReferences into logical containers).

- *Code Generation*: EMF includes a code generation facility that can automatically generate Java classes from the Ecore models.
- *Edit Framework*: EMF Edit is an additional component that automatically generates a user interface for the models. It generates editors, views, and property sheets for your models

1.5. Domain-Specific Modeling Languages

A DSML (Kelly and Tolvanen, 2008) is a specialized and tailored modeling language designed for a specific problem domain, application area, or industry. Unlike general-purpose modeling languages like the UML, which are intended to cover a wide range of modeling scenarios, DSMLs are created with a narrow focus on a particular domain or problem space.

Key characteristics of DSMLs include:

- *Domain Specificity*: DSMLs are specifically crafted to represent concepts, abstractions, and semantics that are relevant and meaningful within a particular domain.
- *Abstraction Level*: DSMLs often provide higher-level abstractions that enable users to express complex domain-specific concepts more concisely and accurately.
- *Expressiveness*: DSMLs are designed to be expressive enough to capture the essential aspects of the domain, using specialized constructs, notations, or modeling patterns.
- *Tool Support*: DSMLs are typically accompanied by dedicated modeling tools or environments, offering features like modeling editors, code generators, and validation mechanisms.
- *Customization*: DSMLs can be customized to adapt to variations within the domain, allowing users to extend or modify them as needed.

A DSL is a programming or modeling language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. DSLs encompass a broader category of specialized languages, including both text-based and modeling languages, while DSMLs specifically refer to modeling languages used in the context of

MDE to represent domain-specific concepts and systems through models. For example, Henshin is a DSML, specifically designed for model transformation, making it domain-specific to that particular aspect of modeling.

The Language Engineering Process ensures that DSMLs are well-suited to their domains and empower domain experts to work effectively.

1.5.1. *Components of a DSML*

A DSML typically consists of several key components (Kelly and Tolvanen, 2008) that work together to provide a specialized modeling and problem-solving environment within a specific domain including:

- *Metamodel*: A metamodel is a high-level, abstract representation that defines the structure and semantics of models within a particular domain. It serves as a blueprint or specification for creating instances of models that adhere to the rules, constraints, and concepts defined by the metamodel. The metamodel defines the abstract syntax of the language or domain it represents. This includes the types of elements (classes), their attributes, relationships (associations), constraints, and the overall structure of

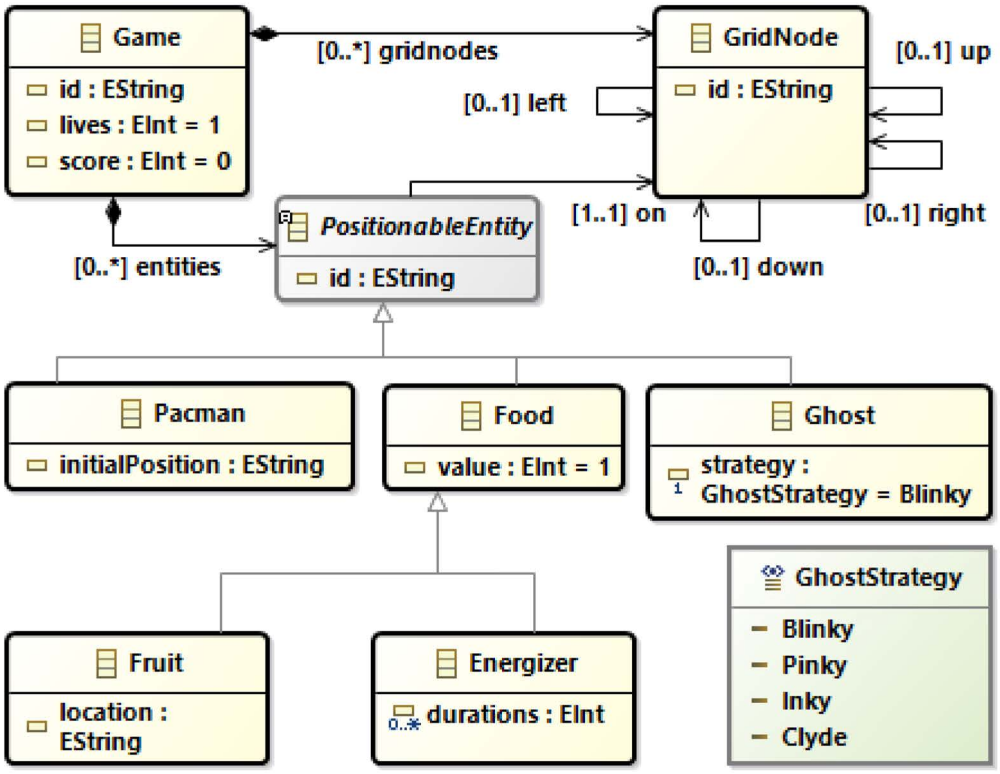


Figure 2.1. Metamodel of DSML

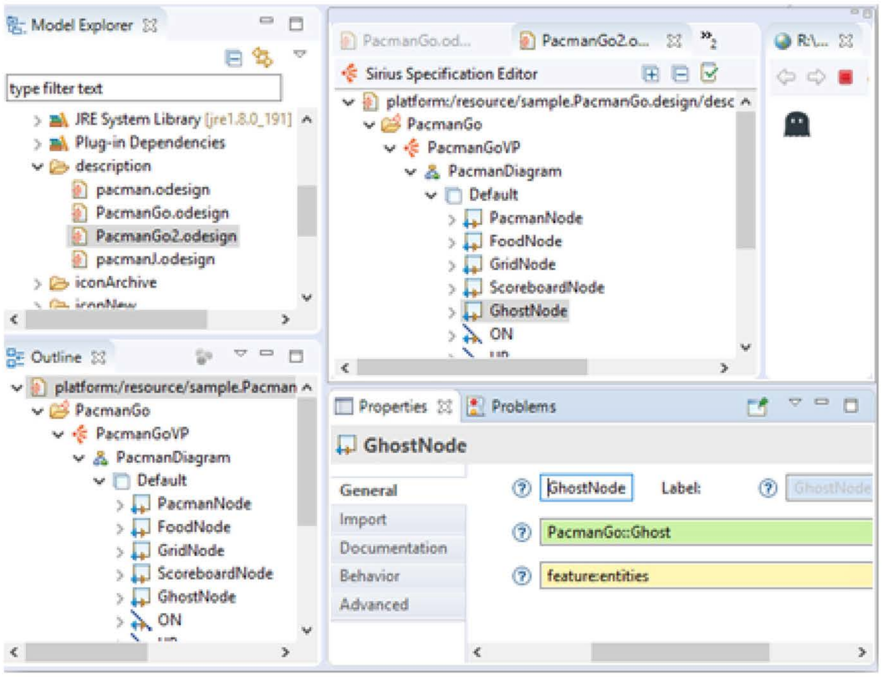


Figure 2.2. Concrete Syntax of DSML

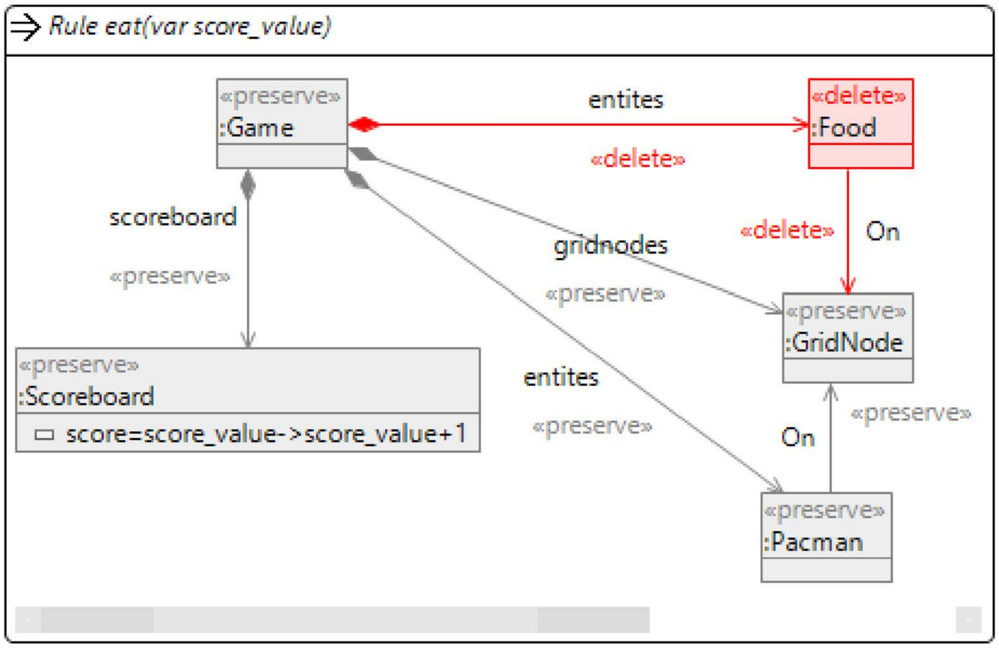


Figure 2.3. Semantics of DSML

valid models. Ecore provides tools and libraries for creating metamodels and models in a standardized way, making it suitable for practical modeling tasks within the Eclipse ecosystem. The abstract syntax is represented in *.ecore* files. Figure 2.1

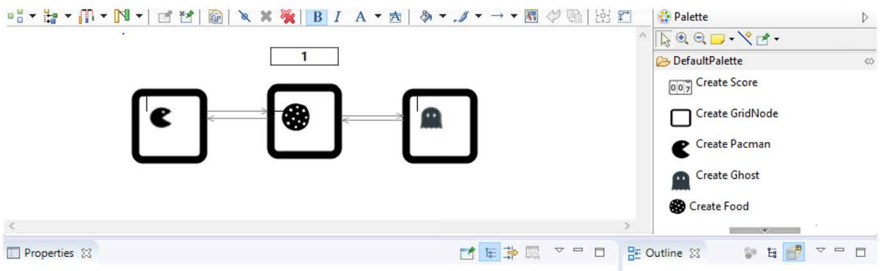


Figure 2.4. Editor of DSML

shows the metamodel for the Pacman game, including Pacman, GridNode, and other metaclasses and associations.

- *Concrete Syntax*: Concrete syntax refers to the specific and tangible representation of models or code in a human-readable format. It focuses on how elements, relationships, and other constructs in a modeling language or DSL are visually or textually represented. Concrete syntax defines how these elements are formatted, displayed, and structured, making it easier for users to interact with and understand models. Figure 2.2 shows the Concrete Syntax for the Pacman game designed using Sirius (Sirius, 2023a). Sirius is a platform for developing and using graphical model editors for any domain. It is based on the Eclipse Platform, and in particular the Eclipse Modeling stack based on EMF.
- *Mappings*: Mappings in this context usually refer to the relationships and transformations between different representations of a language, such as between abstract syntax and concrete syntax. These mappings ensure that a DSL can be both understood by humans in its concrete syntax form and processed by machines in its abstract syntax form. This can involve parsing (converting concrete syntax to abstract syntax) and pretty-printing (converting abstract syntax back to concrete syntax), among other transformations. Figure 2.2 also illustrates the mapping between abstract syntax and concrete syntax of the Pacman game designed using the Viewpoint Specification of Sirius (oDesign file).
- *Semantics*: Semantics of a language define the meaning or behavior of the language's constructs. It describes how the elements of the language should behave when interpreted or executed. The metamodel for semantics often includes formal rules, such as operational semantics or axiomatic semantics, which precisely define how language constructs are executed or evaluated. The semantics can also be derived from the

runtime behavior, which means observing how the language behaves when executed on a computer. For example, if the DSL is for specifying business rules, the semantics could include how those rules are applied to real-world data during execution. Figure 2.3 shows the operational semantics of the *eat* rule using Henshin (Strüber et al, 2017).

- *Editor*: In the context of DSMLs, an editor refers to a software tool or environment that allows users to create, edit, view, and manipulate models conforming to the DSML. This editor is customized and designed specifically for the DSML and its associated metamodel (abstract syntax). The primary purpose of a DSML editor is to provide an intuitive and domain-specific interface for users to work with models in that particular domain. Figure 2.4 shows the editor created for the Pacman game using Sirius.

As discussed, Xtext (Xtext, last accessed 2023) is an open-source framework developed by the Eclipse Foundation for creating DSLs with a strong focus on textual representations (concrete syntax). It provides a way to define the grammar and syntax of the DSML using a custom DSL and then generates an editor, parser, and other necessary components automatically. In contrast, Sirius (Sirius, 2023a), developed by the Eclipse Foundation, is another powerful tool that focuses on creating graphical modeling environments for DSMLs. With Sirius, the DSL engineer can design the custom graphical editors for DSMLs, providing a visual way for domain experts to create, edit, and visualize models.

1.5.2. *Language engineering process in DSML*

The Language Engineering Process (Kelly and Tolvanen, 2008) in DSMLs involves a systematic approach to designing, defining, and implementing a DSML tailored to a specific domain. It encompasses several key steps which we provide five main steps:

(1) *Domain Analysis*: It begin with a comprehensive analysis of the target domain, understanding its concepts, rules, processes, and specific requirements. Domain experts play a crucial role in providing insights during this phase.
(2) *Metamodel Definition*: It needs to define a metamodel based on the domain analysis. The metamodel serves as the abstract syntax of the DSML, specifying types of elements, relationships, and constraints using languages like EMF Ecore.

(3) *Concrete Syntax Design*: It refers to design the concrete syntax of the DSML, specifying visual or textual representations. This includes graphical diagrams, symbols, textual grammar, and syntax rules.
(4) *Editor and Tool Development*: It means to develop tools and editors supporting the DSML, allowing users to create, edit, and manipulate models adhering to the DSML.
(5) *Transformation and Code Generation*: It requires to develop transformation rules or code generation templates for generating code, documentation, or reports from DSML models.

1.6. Model management

Model management encompasses activities such as creating, manipulating, and evolving models (Kolovos et al, 2006a), with key abstractions being models and mappings between them (Bernstein, 2003). Operators like Match, Merge, Diff, Compose, Apply, Copy, ModelGen, and Enumerate facilitate operations such as creating, querying, updating, and deleting models and model elements, enabling the mapping (transformation) of models as bulk objects. The Meta-Object Facility (MOF) stands as a standard model management framework from the OMG (Object Management Group (OMG), last accessed 2023), providing core facilities for defining modeling languages.

The most well-known framework for implementing model management is the EMF (Eclipse Foundation, 2023b). Atlas Model Management Architecture (AMMA), built on ATL (Jouault et al, 2008) atop EMF, serves as a model management framework, offering a virtual machine and infrastructural tools to support model management activities. Epsilon Object Language (EOL), another example, is built on Object Constraint Language (OCL) (Object Management Group (OMG), 2023). EOL can function as a standalone generic model management language or as infrastructure for constructing task-specific languages. The FTG+PM language (Lúcio et al, 2013), developed atop AToMPM (Mannadiar, 2012), comprises the Formalism Transformation Graph (FTG) and its complement, the Process Model (PM). The FTG includes formalisms (nodes) and transformations (edges), describing the languages used at each model development stage. Transformations model development activities, with the control flow and data flow between actions explicitly modeled in the PM.

2. VCS for programming

VCS are software tools that help track changes to files and code over time in a collaborative development environment. They provide mechanisms for recording (versioning), comparing, and merging different versions of files, code, and other artifact types. Versioning refers to maintain a history of changes made to files or code. Each change is recorded as a version, allowing developers to track the evolution of a project over time. Based on how changes are tracked and represented in the VCS, comparisons in VCS are categorized as either state-based or operation-based.

2.1. Differencing

In VCSs, differencing or comparison refers to the process of analyzing and identifying the differences between two or more versions of the same file or set of files (Brosch et al, 2012e). The goal of differencing is to understand how the content has changed over time or between different branches or contributors. The expected result of this process is a clear understanding of what has been added, modified, or deleted in each version, which is typically presented as a set of changes or differences. Based on the type of merging, we have categorized the results into three main types. In the following list, we outline results of each category: Comparisons in VCS are categorized as Line-Based, Model-Based, or Semantic/Domain-Specific based on the type of content being managed and the level of abstraction at which they are performed (Mens, 2002). The outcome of each comparison includes lists of matches, differences, and conflicts.

2.1.1. *Methods*

State-based comparison: VCS use state-based comparison to determine the differences between different versions or states of a file or codebase (Mens, 2002). This comparison method identifies the final state of a file without considering the individual operations that led to that state. State-based comparison is useful for understanding the overall differences between versions. State-based comparison is conceptually simpler to understand and use since they need the final state of the files or models. However, state-based comparison may lose detailed information about individual changes or operations, making it harder to understand the history of modifications.

Operation-based comparison: In contrast, operation-based (change-based) comparison tracks individual changes or operations that occurred between versions (Mens, 2002). This level of granularity allows developers to see exactly what changes were made, such as code additions, deletions, and modifications. Operation-based comparison provides a detailed history of changes, making it easier to understand who made which changes and when. In addition, conflict resolution can be more precise, as the system has knowledge of individual changes and can provide better guidance. However, operation-based comparison depends on special editors to tracks individual changes or operations made to files or models over time.

2.1.2. *Line-based (textual) differencing*

Matching. In line-based textual merging, matching refers to identifying lines or characters in the source files that correspond to each other in the versions being compared. For example, a matching line in two versions of a file may have the same content or a similar content structure.

Differences (Diff). Types of differences in line-based merging include:

- *Text addition*: A line or character that exists in one version but not in another.
- *Text deletion*: A line or character that exists in one version but has been removed in another.
- *Text Modification*: A line or character that exists in both versions but with different content.

Conflicts. Conflicts occur when there are conflicting changes made to the same lines or characters in different versions of a file. Types of conflicts in line-based merging include:

- *Equivalent Conflict*: Occurs when multiple contributors make changes to the same lines or characters, but those changes are semantically equivalent and result in the same final content.
- *Contradicting Conflict*: Occurs when multiple contributors modify the same lines or characters with conflicting content that cannot be automatically merged.

2.2. Merge

Merging is a fundamental process within VCSs that plays a pivotal role in coordinating collaborative software development efforts (Brosch et al, 2012e). It involves the integration of changes made by multiple contributors into a single, coherent version of a codebase or dataset.

Merging ensures that the modifications made by different team members or branches are harmoniously combined, preventing conflicts and maintaining the integrity of the project's history. Merging can be categorized into three main types, each suited to specific use cases and data structures:

2.2.1. *Line-based merging*

Line-based merging, the simplest, is most suitable for text-based files like source code (Mens, 2002). It offers straightforward comparisons at the line or character level. This approach is fast and easy to understand, making it a popular choice for developers working with textual content.

When conflicts are detected, the version control system marks the conflicting sections with special markers and prompts the user to resolve the conflicts manually. Users can employ merge tools or text editors to review and choose between conflicting changes (Git, last accessed 2023; SVN, last accessed 2023), opting to keep changes from the current branch, the incoming branch, or manually edit the content for a custom resolution. After resolving conflicts, the user removes the conflict markers and marks the file as resolved, usually by executing a command such as 'git add' in Git. The merge operation is then completed with a commit, and it's advisable to review the merged file and perform testing, especially in software development, to ensure the successful integration of changes. The line-based merging process ensures that conflicting changes are addressed thoughtfully and that the resulting merge accurately reflects the intended modifications from both branches.

However, its primary limitation is its lack of semantic awareness. It cannot handle domain-specific semantics or complex structural changes effectively, and it struggles when dealing with non-textual data formats (Brosch et al, 2012e).

2.3. Line-Based (textual) merging tools

- *Git (with Diff and Merge Tools).* Git (Git, last accessed 2023), a widely used distributed VCS, offers built-in support for line-based textual merging. Developers can use Git's default merge tools or integrate third-party tools like Beyond Compare or KDiff3 to assist in line-based merging. While Git can support both state-based and operation-based merging, it is often used in a state-based manner. EGit is an

Eclipse-based plugin for version control, specifically designed for working with Git repositories.

- *SVN*. SVN (SVN, last accessed 2023) is a widely used centralized VCS primarily focused on tracking changes to source code files and text-based files. It uses line-based textual merging to manage code versions, compare differences between code changes, and merge changes made by multiple developers. SVN is primarily a state-based VCS.
- *KDiff3*. KDiff3 (kdiff3, 2023) is an open-source diff and merge tool that is especially useful for line-based textual merging. It offers a straightforward interface for comparing and merging text files, highlighting differences, and allowing users to resolve conflicts. is primarily a state-based VCS.

3. Model Comparison

In this section, we discuss model comparison from two different paradigm aspects, including *generic model comparison* and *semantic/domain-specific model comparison*.

3.1. Generic model comparison

The comparison results in generic model comparison include a fine-grained list of model element matches, differences, and conflicts.

3.1.1. *Matching*

In model-based merging, matching involves identifying elements, attributes, or references in different versions of a model that correspond to each other in structure and semantics. Model matching can be: *static identity-based*, which assumes a unique identifier for objects (Kolovos et al, 2009a); *signature-based*, which compares objects based on a dynamic signature calculated from the objects' properties; *similarity-based*, which matches objects based on the weighted similarity of their properties but obviates the model semantics; and *language-specific*, developed ad-hoc for a modeling language and its semantics. For example, EMFCompare is similarity-based but permits defining custom matching algorithms, and UMLDiff (Xing and Stroulia, 2005) is language-specific.

3.1.2. *Differences (Diff)*

Types of differences include (Cicchetti et al, 2008a):

- *Model element addition*: An element, attribute, or reference that exists in one version but not in another.
- *Model element deletion*: An element, attribute, or reference that exists in one version but has been removed in another.
- *Model element modification*: An element, attribute, or reference that exists in both versions but has different content or properties.

3.1.3. *Conflicts*

Conflicts in model-based merging occur when there are conflicting changes made to the same model elements, attributes, or references in different versions (Wieland et al, 2013). Types of conflicts in model-based merging include:

- *Equivalent Conflict*: Occurs when multiple contributors make changes to the same model elements, attributes, or references, but those changes are semantically equivalent and result in the same final model state (Brosch et al, 2012e).
- *Contradicting Conflict*: Occurs when multiple contributors make changes to the same model elements, attributes, or references with conflicting content that cannot be automatically reconciled (Mens et al, 2005).

3.2. Semantic/domain-specific model comparison

The comparison results in semantic/domain-specific model comparison include a combination of fine-grained and coarse-grained lists of model matches, differences, and conflicts.

3.2.1. *Matching*

Depending on the domain or modeling language, matching in semantic/domain-specific merging involves identifying elements or constructs in different versions of a domain-specific artifact that correspond to each other in domain-specific semantics. Pattern matching plays a significant role in semantic/domain-specific merging, commonly employed to identify semantic differences. This involves employing techniques to search for a specific set of model elements, known as a semantic pattern. Graph matching represents a specialized form of pattern matching, where a semantic difference manifests as a match in graphs. In another approach, Schaathun and Rutle (2018) transforms model specifications into RDF graphs by scrutinizing RDF subgraphs conforming to the homomorphisms graph.

3.2.2. *Differences (Diff)*

The types of differences in semantic/domain-specific merging can vary widely, depending on domain-specific semantics, constraints, and rules.

Expansion of generic model versioning approaches.

The expansion of generic model versioning approaches to detect semantic differences involves enhancing the capabilities of versioning techniques to identify and comprehend changes at a semantic level within models. Generic model versioning typically deals with managing different versions of models, tracking changes, and facilitating collaboration among multiple contributors. However, detecting semantic differences goes beyond mere syntactic changes and aims to capture alterations that have a meaningful impact on the semantics or interpretation of the models.

EMF Compare (EMF Compare, accessed August 2023) provides various extension points allowing the integration of custom behaviors throughout different stages of the model versioning process. Currently, specific editing dynamics are handled through manually crafted code that customizes generic behaviors within EMF Compare. While this serves as an initial solution, it proves to be neither scalable nor sustainable (Koegel and Langer, 2015; Zadahmad et al, 2022). Addressing numerous editing dynamics in a modeling language manually within EMF Compare becomes burdensome and may result in inconsistencies, particularly as these dynamics evolve over time.

Moreover, editing dynamics can significantly vary from one modeling language to another or based on the application of a modeling language in different organizations or projects. Domain-specific modeling languages inherently possess unique editing dynamics, derived from the purpose and pragmatics of the respective language. The use of UML Profiles to define domain-specific modeling languages (Sharbaf and Zamani, 2017), adds an additional layer of complexity (Koegel and Langer, 2015), as the application of a UML Profile can substantially impact how a model is edited.

Even though the knowledge about editing dynamics is typically implemented in modeling editors, it lacks explicit availability in a reusable manner. Consequently, this knowledge needs to be redundantly re-implemented in model versioning systems, creating a gap between modeling editors and model versioning systems (Koegel and Langer, 2015).

Schipper et al (2009) expanded EMFCompare to illustrate schematic differences in diagrams. However, their extension only allows for visualizing atomic changes and doesn't support coarse-grained modifications.

Semantic lifting.

Semantic lifting in the context of domain-specific model differencing refers to the process of abstracting or elevating the comparison and merging operations to a higher level of semantic representation. This is done to enable a more meaningful and context-aware analysis of changes within models that adhere to a specific domain or modeling language. Semantic lifting aims to improve the accuracy of model differencing by reducing false positives and false negatives. It helps in distinguishing between changes that are semantically significant within the domain and those that are merely syntactic. Kehrer et al (2011) employs a semantic lifting approach to address the potential challenges associated with comprehending low-level differences provided by generic comparison tools. This method elevates these differences to the level of editing operations. To identify editing processes, the approach involves grouping relevant low-level changes. Subsequently, these groupings are utilized to describe low-level differences as models. An essential aspect of this approach is the automatic generation of rules from the rule-based description of the editing operations.

In the pursuit of semantic lifting for domain-specific model differencing, domain experts play a critical role in providing specialized knowledge and insights. To initiate this process, experts are tasked with articulating domain-specific semantics and concepts, defining the meanings and relationships of elements within the models. Detailed information about the modeling language employed in the domain is essential, encompassing aspects such as syntax, structure, and modeling rules. Furthermore, domain experts identify and communicate semantic constraints or rules that govern the models, ensuring that the lifting process aligns seamlessly with the expected behavior of the models.

Additionally, domain experts contribute significantly to determining the significance of changes within the domain. Their input guides the specification of which alterations should be emphasized during the semantic lifting process. Contextual knowledge about the domain and the models' purpose is invaluable for making informed decisions during semantic lifting, allowing for a nuanced interpretation of semantics based on real-world implications.

In terms of techniques, domain experts may be involved in the development and refinement of algorithms for semantic lifting. This may include specifying rules for grouping relevant low-level changes, a crucial step in identifying editing processes. Techniques for the automatic generation of rules from rule-based descriptions of editing operations are essential aspects of this approach.

Furthermore, domain experts can contribute by providing specific use case scenarios or examples that illustrate how changes in the models may impact the overall system or application. These scenarios help guide the semantic lifting process, providing context for the development of techniques.

As tools evolve for semantic lifting, continuous feedback from domain experts is crucial for refining techniques and algorithms. This collaborative effort ensures that the lifting process remains aligned with the evolving needs of the domain. Finally, domain experts may play a role in validating the results of semantic lifting, ensuring that the elevated semantics align with their understanding of the domain. This feedback loop contributes to ongoing improvements in the accuracy and relevance of the lifting process. The collaborative synergy between computer scientists and domain experts, coupled with the application of specialized techniques, is fundamental to achieving an effective and meaningful lifting of low-level differences to a higher semantic level in the domain-specific context.

Two-way model differencing.

Two-way model differencing involves comparing two versions of a model to determine the differences between them. Its primary goal is to detect and highlight changes, such as additions, deletions, and modifications, that have occurred between the two versions. However, the main shortcoming of two-way differencing is the lack of a common ancestor, which can make conflict resolution more challenging. This limitation results in a limited understanding of how changes relate to the overall development history.

For instance, two-way comparison may not effectively identify conflicts, especially when the same element or piece of information has been modified in both versions being compared. Consequently, it might not provide sufficient information to determine how to merge such parallel changes. Moreover, two-way differencing often interprets renames and moves as deletions in one version and additions in another, leading to potentially complex conflict resolution scenarios. As a result, two-way comparison may require more manual intervention and involve complex merge strategies.

In the realm of DSM, Cicchetti et al. (Cicchetti et al, 2007) propose an approach to represent model differences that is metamodel independent and agnostic of the difference calculation method. Specifically, given two models conforming to the same metamodel, their difference is expressed as another model that conforms to a new metamodel. This new metamodel is derived from the original one by a transformation and allows representing model changes (additions, deletions, and changes). Such difference models induce transformations to translate from one model version to the other and can be composed. The work only works at the abstract syntax level.

(Kühne et al, 2009) introduce an approach based on graph transformation rule patterns to express differences in a domain-specific way. The metamodel of the patterns is generated by transforming the metamodel of the input/output DSLs: relaxing cardinalities, adding transformation-specific attributes and other concepts, and modifying attribute types.

Since low-level differences returned by generic comparison tools may be difficult to understand, Kehrer et al. (Kehrer et al, 2011) perform a semantic lifting of such differences to the level of editing operations. For this purpose, low-level differences are represented as models, so that the identification of editing operations consists of finding groups of related low-level changes. This search is performed by rules that are automatically derived from the rule-based specification of the editing operations. However, semantic lifting in this approach only deals with the abstract syntax of models.

Semantic lifting approaches such as (García et al, 2013; Vermolen et al, 2012) identify complex change patterns from low-level changes involved in a metamodel evolution. However, these patterns are generic and predefined. We need an approach allowing the DSL engineer to define the semantic differencing rules using a domain-specific editor.

Three-way Model differencing.

Three-way model differencing involves comparing three versions of a model, including branch Version one (left version, ours), branch Version two (right version, theirs), and a common ancestor (base version) for the other two versions. Its purpose is to identify, analyze, and highlight differences, conflicts, and similarities among these versions.

Compared to two-way differencing, three-way differencing offers a more comprehensive view of changes. It excels in detecting conflicts, especially when both branches have made conflicting changes to the same elements or properties. Consequently, it allows for the utilization of more sophisticated merging algorithms. Three-way differencing can automatically merge changes that do not conflict.

However, it is important to note that three-way differencing still requires user intervention to resolve contradicting conflicts. Additionally, it is more complex to implement compared to two-way differencing, given the need to manage three versions and their relationships

In the realm of DSM, Schipper et al (2009) extended EMFCompare to depict schematic differences in diagrams.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 conduct 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 (2013a) post-processes atomic operations 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.

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.

Maoz et al (2011b) map models to a formally defined semantic domain in order to reason about the differences.

3.2.3. *Conflict Management*

Conflicts in semantic/domain-specific merging depend on domain-specific semantics, constraints, and rules (Brosch et al, 2010a). These conflicts may be resolved as equivalent conflicts (changes are semantically equivalent) or contradicting conflicts (changes conflict and cannot be automatically merged) (Altmanninger and Pierantonio, 2011). Constraint violation, pattern matching, change overlapping, and formal methods are the common categories of conflict detection techniques (Sharbaf et al, 2022b). Each approach may simultaneously benefit more than one technique to detect conflicts. The constraint violation techniques can check model and metamodel constraints (Tröls et al, 2021), whereas the change overlapping techniques comprise contradicting changes and equivalent changes (Sabetzadeh and Easterbrook, 2006). On the other hand, pattern matching can be utilized to find more complex conflicts involving multiple elements from different models (Fritsche et al, 2020). Finally, formal methods can be used to find conflicts through rigorous mathematical approaches (Zerrouk et al, 2018).

Conflict Detection.

In the context of VCS a conflict refers to a situation where two or more changes made by different contributors or branches are in conflict with each other and cannot be automatically merged without human intervention (Booch et al, 2000). In model-based comparison, conflicts can manifest in various ways, leading to challenges in merging changes made by different contributors. Structural conflicts (syntactic or fine-grained) occur when multiple contributors modify the same structural elements within a model. For example, if one developer deletes a class that another developer's class diagram references as a superclass, a dangling reference conflict arises. Another example is when two developers concurrently modify the attributes of a class, with one adding a new attribute and the other renaming an existing one, causing an overlapping changes conflict.

The conflict arises because it is unclear whether the modification should apply to the deleted element or if the deletion should override the modification. Similarly, if one branch modifies an element or property in a way that is inconsistent with changes made in another branch, a conflict occurs. For example, if one branch modifies the cardinality of a reference from "0..1" to "1" while another branch changes it to "0..n," an attribute or reference conflict arises.

Semantic conflicts (coarse-grained or domain-specific) (Brosch et al, 2012e), occur when changes to the model affect its semantics or behavior, potentially introducing inconsistencies or non-compliance with domain-specific rules. These conflicts are often complex to resolve, as they demand a deep understanding of the model's intended behavior and the domain-specific rules governing it. In a model-based environment, violations of constraints conflict may arise when changes made in different branches lead to violations of domain-specific constraints, model integrity rules, or behavioral constraints. For example, if one developer changes the state transition rules of a state machine model, and another developer modifies a related constraint that contradicts the new rules, a constraints violation conflict occurs.

Another concepts is equivalent conflicts that happen when multiple contributors make changes to the same model elements, attributes, or references, but those changes are semantically equivalent and result in the same final model state. For example, two modelers independently rename a set of classes to more descriptive names, and their changes are semantically equivalent because they result in the same model structure.

To detect conflicts, four main approaches have been proposed, as discussed by Sharbaf and colleagues (Sharbaf et al, 2022b). These approaches include conflict pattern matching, which involves identifying model fragments representing conflict patterns; constraint violation checking, which assesses adherence to well-formedness rules for models and metamodels; change overlapping checking, which identifies equivalent conflicts; and formal methods, which rely on mathematical and formal logical principles.

The conflict pattern language introduced by Sharbaf et al (2020) is used to express conflicts in different modeling languages written in Epsilon model validation language (EVL) (Kolovos et al, 2009b). However, it only detects conflicting semantics and ignores non-conflicting semantics. Whereas, DSMCompare identifies all semantic differences, offering users a comprehensive overview of changes at a semantic level. This aids users in distinguishing the intended resolution when resolving conflicting semantic differences. Moreover, in our approach, we find semantic differences and investigate them for semantic conflicts. 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. Morover, their patterns are general and predefined, since they do not provide users with a domain-specific mechanism to define new rules. Yet, they resemble the rules in our method. Therefore, it lack flexibility for diverse conflicts across modeling languages, making managing semantic conflicts a challenge in model merging.

An approach in (Altmanninger et al, 2010) focuses on modeling language semantics and can detect semantic conflict. However, it relies on defining semantic views and representing models to introduce specific semantic aspects (Altmanninger, 2007). Other methods (e.g., (Sharbaf and Zamani, 2020; Dam et al, 2016)) mainly address static semantic conflicts and are tailored to specific modeling languages, limiting their use.

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 constraint violation checking. However, their approach disregards the effect of syntactic modifications on the semantics of the model as explained by Kautz and Rumpe (2018).

AMOR (Altmanninger et al, 2008a) provides a change overlapping checking using critical pair analysis examine whether a pair of operations, one of which contains a glue element, can be merged into a single operation. (Mougenot et al, 2009) propose an approach that utilizes description logics and logical inference techniques for automating conflict detection. However, it might have limitations in expressing complex domain-specific concepts and relationships. As a result, it lacks a mechanism for easily incorporating and adapting to domain-specific rules.

Brosch et al (2012b) create a separate difference 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 and constraint violations. However, they are specific to UML class diagrams.

Sharbaf and Zamani (2020) use UML profiles to model a conflicting case by defining examples of the conflict parties as a pattern. They also highlight the conflicts using different colors. However, their approach is only appropriate for UML models.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 and Kolovos, 2022). However, when using PEACEMAKER, DSL users must reason about differences and conflicts at the abstract syntax level rather than using the concrete syntax.

Conflict awareness.

Conflict awareness involves continuously monitoring changes made by different contributors to a shared model. It identifies situations where changes overlap, contradict, or are otherwise incompatible with each other.

When a conflict is detected, the VCS system raises a notification to inform the relevant users about the conflict. This notification typically includes details about the conflicting changes, such as which elements or attributes are affected, who made the conflicting changes, and the specific nature of the conflict (e.g., structural or semantic). For example, (Tröls et al, 2019) uses modeling approach to show warning messages.

Conflict awareness often provides a visual or textual representation of the conflicting changes within the modeling environment. This representation helps users understand the nature of the conflict and the specific areas of the model that are affected. For instance, (Bartelt and Schindler, 2010) highlights the involved model elements when a conflict item is selected. While, (Baqasah et al, 2014) introduces a conflict view, which opens a new window providing additional details about conflicts, including descriptions and links to the conflicting elements.

Conflict Resolution.

Conflict resolution is the systematic approach of resolving conflicting changes made by multiple contributors when integrating different branches or versions of a shared resource. Conflict resolution can be categorized into several types, including manual, semi-automatic, and automatic methods, each with its strategies and components:

Manual.

Manual conflict resolution involves human intervention for conflict resolution. In this process, users actively participate by making decisions based on their understanding of the model and the intentions behind the changes. They have several strategies at their disposal, including selecting one version over the other (Keep left / Keep right), applying changes from one version first and then the other (Apply left then right / Apply right then left), implementing custom resolutions (modifying conflicting sections to achieve desired outcomes), or discarding all changes (Sharbaf et al, 2022a; Brosch et al, 2012e). For instance, in the work of (de la Vega and Kolovos, 2022), users are presented with the options to retain either the left or right version or to take no action in certain conflicting cases. The work also provides the capability to automate conflict resolution by automatically merging and removing conflicting elements.

Semi-automatic.

Semi-automatic conflict resolution combines human judgment with automated assistance, striking a balance between user involvement and streamlined conflict resolution. In this approach, the VCS offers tools and suggestions to aid users in the process. Users play an active role by reviewing and approving suggested resolutions, with the flexibility to make adjustments as needed (Debreceni et al, 2016).

Semi-automatic resolution strategies encompass various techniques, including the automatic merging of non-conflicting changes, the identification of potential conflicts with accompanying suggestions for resolution, and interactive conflict resolution interfaces. In the work of (Kuiter et al, 2021), a conflict resolution process based on negotiation and voting among collaborators is employed to identify their preferred modifications. For instance, in the study by (Debreceni et al, 2016), possible resolution candidates are computed for each conflict, allowing users to select the most suitable one.

Automatic.

Automatic conflict resolution is designed to streamline the conflict resolution process by minimizing direct user intervention. In this approach, VCS employ predefined rules, algorithms, or heuristics to automatically merge changes whenever possible. Typically, developers only need to step in when conflicts cannot be resolved automatically or when the VCS requires input to make a decision.

Automatic resolution components may include several features and strategies. For instance, some VCS systems, like Git (Git, last accessed 2023), permit users to define conflict resolution priority rules. Users can specify that changes to specific files or sections of code should consistently take precedence in case of conflicts, guiding the automatic resolution process. Organizations can also establish conflict resolution policies that outline how particular types of conflicts should be automatically handled. These policies can be based on coding standards, best practices, or project-specific requirements. For instance, when merging changes from the *development* branch into the *release* branch, policies may prioritize changes from the *release* branch for critical files associated with the *release*.

Some methodologies (e.g., (Wieland et al, 2013) and (Tröls et al, 2019)) fully support manual conflict resolution, involving users in making the final choices among suggestions during the resolution phase. However, a lack of insight into the intentions each user had during modeling can result in overlooked support for established requirements or even give rise to new conflicts (Chong et al, 2016; Brosch et al, 2012d).

Moreover, tools such as flake8 (flake8, 2023) in the context of Python can be employed to validate and format code according to predefined standards. Platforms like GitHub Actions (githubActions, 2023) and GitLab CI/CD (gitlabCICD, 2023) offer automation capabilities that can be integrated into the conflict resolution process, enhancing its efficiency.

Additionally, other approaches involve operational transformation or the application of predefined conflict resolution patterns. For instance, (Nicolaescu et al, 2018) utilizes the operational transformation algorithm to generate new operations consistently applicable across all versions for each conflict. Conversely, (Rossini et al, 2018) applies predefined conflict resolution patterns.

Certain approaches (Hachemi and Ahmed-Nacer, 2020; Fritsche et al, 2020) perform resolution for limited conflict scenarios. DSL engineers can expand the repository of rules and tailor existing conflict specifications and resolution components. These rules are then translated into equivalent Henshin rules (Strüber et al, 2017), enabling their utilization by external tools, such as machine learning applications, for enhanced effectiveness (Eisenberg et al, 2021).

4. Model Merge

Merging is a fundamental process within VCSs that plays a pivotal role in coordinating collaborative software development efforts (Brosch et al, 2012e). It involves the integration of changes made by multiple contributors into a single, coherent version of a codebase or dataset.

Merging ensures that the modifications made by different team members or branches are harmoniously combined, preventing conflicts and maintaining the integrity of the project's history.

Model merging, in the context of MDE, refers to the process of integrating changes and updates from multiple versions of models into a single coherent model (Mens, 2002). Model-based merging often involves a high level of semantic awareness in addition to the structural aspects of models, such as classes, associations, and attributes. It considers the meaning and semantics of changes to ensure that the merged model remains consistent and adheres to domain-specific knowledge and constraints.

If a change of structural or semantic types appears in more than one branch of the model (equivalent changes), only one copy of it should be included in the merged model. This property is known as non-redundancy. If a change of structural or semantic types contradicts other branches of the model (contradicting changes), the conflict should be resolved before the merge.

4.1. paradigm

Merging can be categorized into three main types, each suited to specific use cases and data structures:

4.1.1. *Line-Based merging*

Line-based merging, the simplest, is most suitable for text-based files like source code (Mens, 2002). It offers straightforward comparisons at the line or character level. This approach is fast and easy to understand, making it a popular choice for developers working with textual content.

When conflicts are detected, the version control system marks the conflicting sections with special markers and prompts the user to resolve the conflicts manually. Users can employ merge tools or text editors to review and choose between conflicting changes (Git, last accessed 2023; SVN, last accessed 2023), opting to keep changes from the current branch, the incoming branch, or manually edit the content for a custom resolution. After resolving conflicts, the user removes the conflict markers and marks the file as resolved, usually by executing a command such as 'git add' in Git. The merge operation is then completed with a commit, and it's advisable to review the merged file and perform testing, especially in software development, to ensure the successful integration of changes. The line-based merging process ensures that conflicting changes are addressed thoughtfully and that the resulting merge accurately reflects the intended modifications from both branches.

However, its primary limitation is its lack of semantic awareness. It cannot handle domain-specific semantics or complex structural changes effectively, and it struggles when dealing with non-textual data formats (Brosch et al, 2012e).

4.1.2. *Model-Based Merging*

Model-based merging operates at a higher level of abstraction, making it well-suited for structured data represented as models, which is common in MDE. This approach excels in understanding the abstract-syntax of the model, making it suitable for complex structural changes. Model-Based Merging provides a detailed history of changes and supports precise conflict resolution, making it valuable for MDE scenarios. However, it struggle with semantic conflicts and can be more complex to implement and understand, particularly for domain experts who are not familiar with the underlying modeling languages.

Upon conflict detection, the system marks the conflicting sections within the models, often using specialized annotations. A three-way comparison is conducted, considering the common ancestor, changes in the current branch, and changes in the incoming branch. Users are presented with a graphical user interface to navigate and visualize the abstract syntax of models (EMF Compare, accessed August 2023; Koegel and Helming, 2010b), with conflicts highlighted for resolution. Users make decisions on merging conflicting changes, such as accepting modifications from one branch, the other branch, or manually resolving discrepancies. The system then applies these decisions to create a merged model, potentially automating the merging of non-conflicting changes. After resolving conflicts, the system updates the models and removes any conflict markers or annotations. Users signal to the system that conflicts are resolved, finalizing the merge operation. In a version control context, users may commit the merged model. Subsequently, users review the merged model to ensure accurate conflict resolution, and additional testing or validation steps may be undertaken to confirm the integrity of the integrated changes. This process ensures that conflicts in higher-level abstractions are addressed systematically, reflecting the intended modifications across shared entities in the model, and can be adapted to various model-based merging tools.

The three-way merging of models has been approached through various methods. Bartelt (2008) proposed a generic three-way merge at a low level of abstraction, potentially resulting in inconsistent models concerning the metamodel and underlying meta-metamodel. To address this, a post-processing phase automatically detects and interactively resolves inconsistencies. Blanc et al (2008) presents a general strategy for identifying inconsistencies in models based on pre-defined consistency rules and Prolog-based first-order logic. However, it does not consider the identification of semantic equivalences in cases of syntactic redundancies.

In contrast, Kolovos et al (2006b) introduced the EML, a rule-based language for two-way model merging, providing customization of merge logic across diverse metamodels. However, EML lacks support for three-way merging and does not handle conflict management for several types of conflicts. Westfechtel (2014) offered a formal approach to three-way merging using set theory and predicate logic for Ecore models. This approach specifies merge rules managing additions, deletions, renaming, and movements of model elements, ensuring a well-formed merged model. While it effectively detects and resolves conflicting modifications, it limits conflict detection to preconditions in subgraph transformation rules.

On the other hand, Rossini et al (2010) explored category theory for three-way merging, representing models as graphs. Their approach involves creating a union graph with elements from all input versions. Conflicts are detected using generic rules that can be augmented with specific rules considering metamodel constraints. However, this method only identifies conflicts and does not provide resolution, and the merge process stops upon detecting conflicts. It also operates at a more general level and would require adaptation for merging EMF models.

4.1.3. *Domain-Specific Merging*

Domain-specific merging is a specialized approach designed for contexts where data and models are inherently domain-specific (Sharbaf et al, 2022b; Brosch et al, 2012e). It focuses on preserving the relevance and validity of artifacts within a specific domain. This approach can incorporate features of model-based merging, providing a high level of precision in preserving domain-specific context. However, domain-specific merging necessitates domain expertise for implementation and may introduce additional complexity based on the specific requirements of the domain.

Semantic or domain-specific model merging is a specialized process for harmonizing changes within models that represent complex systems . In this context, the merging process is guided by the unique semantics, rules, and structures inherent in the domain-specific models (?). The first step involves the detection of conflicts, wherein specific rules and heuristics tailored to the characteristics of the domain are applied to identify discrepancies. Conflict resolution strategies are then crafted or applied, leveraging domain-specific knowledge to align changes based on the semantic meaning of entities within the models (Sharbaf et al, 2022b; Brosch et al, 2012c). The merging tools employed are purpose-built for the nuances of the domain-specific models, often offering visualizations that aid users in understanding and resolving conflicts within the context of the model structure.

The user interface for resolving conflicts is customized to the specifics of the domain, offering options that align with the semantics of model entities. Manual resolution may be required, and users are empowered to make decisions that reflect the intended semantics of the models. The validation and testing phase is critical, involving domain-specific tests to ensure the correctness of the merged model in accordance with the expected semantics (Sharbaf et al, 2022b). A feedback mechanism is established, allowing domain experts to provide insights and corrections based on their specialized knowledge. This iterative process of user feedback contributes to the continuous improvement of domain-specific model merging algorithms and strategies over time, enhancing their alignment with the unique requirements of the specific domain. Ultimately, semantic/domain-specific model merging ensures that the combined models accurately reflect the intended semantics and structures within the targeted domain.

The proposal by Altmanninger et al (2010) aims to enhance the specification of modeling language semantics to facilitate accurate conflict detection. They highlight the importance of considering both syntax and semantics in conflict resolution, showcasing instances where models in syntactic conflict can be merged seamlessly based on their semantics, and vice versa. This tool enables rule definition for conflict detection and resolution but does not consider syntactically different but semantically equivalent parts in concurrently modified models.

Cicchetti et al (2008b) introduces a domain-specific language for specifying both syntactic and semantic conflicts using the difference model, which describes modifications in subsequent versions. However, this formalism cannot specify semantic equivalent conflicts. In the work of Kaufmann et al (2010), conflict detection characteristics are enhanced by user-defined activities, and collaborative conflict resolution characteristics are provided. However, it lacks a formal treatment of conflict detection and resolution.

The work by Altmanninger (2007) focuses on making semantic aspects of modeling languages explicit using semantic views. While it complements our work by aiming for more precise conflict detection through semantic views, it cannot detect semantic equivalent conflicts related to refactoring or complex equivalent concepts. Additionally, it eliminates syntactic redundancies but does not cover syntactic conflicts resulting from atomic and composite change operations. In the formal approach proposed by Taentzer et al (2010), conflicts are described based on graph transformations and the theory of categories. While they identify operation-based and state-based conflicts, their approach lacks consideration for the detection of operation sequences, semantic equivalent modifications, and resolution of detected conflicts.

4.2. Merging methods

There are two main methods for merging including state-based merging and operation-based merging. The categorization is based on whether the merging strategy centers around comparing and merging entire states (state-based) or tracking and applying individual changes (operation-based).

4.2.1. *State-based merging*

State-based merging uses the match, difference, and equivalent list produced as a result of state-based differencing. It typically doesn't consider the history of changes but focuses on the final content. Advantage is simplicity and ease of use for scenarios where tracking individual operations is not necessary. However, the merge may lead to conflicts when multiple contributors make changes to the same parts of the model since it doesn't provide detailed insights into how conflicts arose.

4.2.2. *Operation-based merging*

Operation-based merging considers the history of operations applied to models. It tracks the sequence of changes made by contributors, allowing for precise conflict detection and resolution. It tracks the sequence of changes made by contributors, allowing for precise conflict detection and resolution. Provides detailed insights into the evolution of models, allowing for more fine-grained conflict resolution. Well-suited for complex collaborative scenarios. However, It may be more complex to implement and use than state-based merging since it requires tracking and managing a history of operations.

4.3. Merging techniques

Raw merge, two-way merge, and three-way merge Fig. 2.5 are three distinct techniques used to integrate different versions of a model (Sharbaf et al, 2022b; Mens, 2002).

4.3.1. *Raw Merge*

refers to a straightforward approach of combining changes made to models without performing sophisticated conflict resolution or considering the semantics of the models. In raw differencing, there may not be explicit versions or a common ancestor. The process involves comparing two sets of changes, often represented as patches or diffs, and applying them to a base or original version to create a successive version. It may be employed in very simple cases where structural and semantic conflicts are unlikely, such as merging separate model fragments with minimal dependencies.

4.3.2. *Two-way Merge*

in model-based merging involves using the result produced by two-way differencing in order to integrate the changes of two models. In model-based two-way merge, the tool or system analyzes the differences between the base version (V1 or V2) and the modified

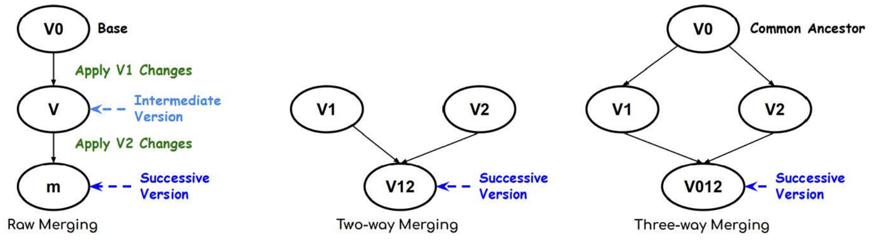


Figure 2.5. Merging Techniques

version (V2 or V1) (Mens, 2002). It considers structural and semantic conflicts and attempts to automatically reconcile them based on predefined rules or user guidance. Two-way merge is useful in model-based merging scenarios where a base version and one modified version need to be integrated. It helps ensure that changes made in the modified version align correctly with the base version, particularly when there is a shared history of changes.

4.3.3. *Three-way Merge*

in model-based merging involves using the result produced by three-way differencing in order to integrate the successive model. Model-based three-way merge analyzes the differences between the common ancestor and both modified versions(Mens, 2002). It identifies structural and semantic conflicts and provides a structured approach to conflict resolution. Developers can review and manually resolve conflicts when necessary. Three-way merge is highly effective in merging non-conflicting fine-grained model-based merging for collaborative MDE projects. It ensures that changes made by multiple contributors or branches are integrated systematically, maintaining the integrity and consistency of the models.

5. Tools

Based on the type of merging, we have categorized merging tools into three main types. In the following list, we outline examples of (commercial) tools for each category.

5.1. Focus

In this section, we focus on existing generic model-based merging tools and semantic/domain-specific merging tools.

5.1.1. *Model-Based Merging*

In this section, we discuss three popular model-based merging tools: *EMFCompare*, *EMFStore*, and the *Epsilon Merging Language (EML)*.

- *EMFCompare.* EMFCompare (EMF Compare, accessed August 2023) is an Eclipse-based tool specifically designed for model-based merging in the context of the Eclipse Modeling Framework (EMF). It allows users to compare and merge EMF models, including UML diagrams, Ecore models, and other EMF-based artifacts. is primarily a state-based VCS. CDO (Connected Data Objects) provides a powerful version control mechanism for managing models. However, it doesn't include a built-in model comparison and merging engine as robust as EMF Compare's. Instead, it relies on external tools like EMFCompare for these specific tasks.

- *EMFStore.* EMFStore (Koegel and Helming, 2010b) is a VCS designed for models and model-based artifacts created using the Eclipse Modeling Framework (EMF). It supports model-based merging for various modeling languages and metamodels, making it a suitable choice for MDE projects. EMFStore is an operation-based SVN.
- *Epsilon Merging Language (EML).* EML (epsilon, 2023) is a domain-specific language designed for model merging tasks. It can be used in conjunction with Epsilon's model management tools to perform model-based merging on various modeling languages and metamodels. EML is primarily a state-based merging approach.

5.1.2. *Semantic/Domain-Specific Merging*

In this section, we discuss two semantic merging tools: *MetaEdit+* and *AMOR.*

- *MetaEdit+.* MetaEdit+ (Kelly, 2018) is a modeling and DSM environment that includes semantic merging capabilities. It enables domain-specific merging for models created using the MetaEdit+ modeling language. MetaEdit+ is primarily a state-based VCS.
- *AMOR.* AMOR (Altmanninger et al, 2008b) is a tool used for managing versions and changes in the context of MDE. It specializes in model-based merging, allowing users to merge changes made to models and diagrams, particularly in the DSM context.AMOR is primarily a state-based VCS.

5.2. Techniques

Model-based versioning tools, like text-based tools, include two merging techniques: two-way versioning and three-way versioning.

5.2.1. *Two-Way VCSs for models*

Even though models are frequently persisted as text files, the use of traditional text-based VCSs is suboptimal, as we have argued in the introduction. This way, several model repositories with support for version control have been proposed along the years (Altmanninger et al, 2009). The ModelCVS (Kappel et al, 2006) and the AMOR projects (Altmanninger et al, 2008a) proposed dedicated VCSs for models with sophisticated functionalities, like a recommender of possible resolutions for model conflicts (Brosch et al, 2010d). In this setting, DSMCompare could be useful to help understand better the differences between the models, before choosing a resolution strategy.

The model repository of Espinazo-Pagán and García-Molina (Espinazo-Pagán and García-Molina, 2010) uses a MySQL database for storage, and a special encoding of model versions to improve efficiency. For a better performance, the authors later proposed the use of NoSQL databases for persistence (Espinazo-Pagán et al, 2011). EMFStore (Koegel and Helming, 2010b) and CDO (CDO Model repository, accessed August 2023) are well-known model repositories for EMF, which support collaborative editing and versioning of models. DSMCompare could be used atop these repositories to enable the visualization of (semantic) diffs using the graphical concrete syntax of the DSL.

Commercial modeling tools feature different levels of versioning and model differencing capabilities. LabView has a built-in revision control system that allows to compare two different models (LabView, 2023) and provides a text-based results. MetaEdit+ (Kelly et al, 1996) features a version control mechanism called Smart Model Versioning (SMV) (Kelly, 2018), which allows comparing models – graphically, textually or by means of a tree – and storing them on any major version control system such as Git. Depending on the configurability of SMV in MetaEdit+, users may have the ability to customize conflict resolution strategies in fine-grained difference level to align with their specific modeling practices and collaboration workflows. MPS (MPS, last accessed 2023b) integrates with Git and Subversion and provides some capabilities for viewing model differences, in a textual way (MPS, last accessed 2023a). Simulink supports comparing models and highlighting the differences in the original models. Simulink uses a scoring algorithm to determine if two model elements are a match (Simulink, 2023). Similarly, SystemWeaver (SystemWeaver, 2023) provides versioning capabilities at the model element level. This way, users can compare an element, view its history, and replace one version of an element with another. While these tools offer different ways to diff models, they are typically fixed and do not support extensive customization or provide support for customization at a fine-grained level. Instead, our approach could be valuable here to provide domain-specific, customizable visualizations of the model differences, in a graphical way.

Our approach is based on Eclipse Modeling Framework (EMF). This is a relevant technology, since Eclipse is widely used in MDE research and many companies use Eclipse and EMF tools (Akdur et al, 2018). Large companies such as IBM are spearheading MDE through EMF (Mohagheghi et al, 2013).

Model differencing and collaborative modeling can lead to clones and duplicates. Some approaches have addressed this problem. Störrle has developed a number of heuristics and algorithms to detect clones in models (Störrle, 2010, 2017). Babur *et al.* (Önder Babur et al, 2019) leveraged natural language processing, feature extraction and clustering techniques to detect clones in models. We have not focused on detecting model clones in our approach, which is left as future work.

5.2.2. *Three-way VCSs for models*

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[1] provides controlled access to all artifacts, simple configuration management, and a mechanism to prevent version conflicts in this manner. Obeo Designer[2] and CDO can integrate with EMFCompare to provide a generic model-based versioning service. Smart Model Versioning[3], a version control tool included in MetaEdit+ (Kelly et al, 1996), 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

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

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.

5.3. Visualization

Gleicher (Gleicher, 2018) provides general guidelines for visualizing comparisons. For many different domains, comparing artifacts is a common task and visualizing the comparison often helps. Generally, the visual comparison is displayed using juxtaposition (*e.g.,* as EMFCompare), superposition, or explicit encoding.

5.3.1. *Juxtaposition*

Juxtaposition in visualization refers to the placement or arrangement of elements in close proximity to one another to create a contrast or comparison (Gleicher et al, 2011). In the context of model comparison tools, it involves the comparison of different versions or instances of a model to highlight changes, differences, or similarities between them. Juxtaposition improve model comparison visualization in different aspects:

- *Element-Level Comparison:* Juxtaposition occurs when the tool identifies elements that exist in one version of the model but not in another, or when the properties of elements differ.
- *Structural Differences Highlighting:* The tool visually highlights structural differences between models. This can include added, deleted, or modified elements, making it easy for users to identify the changes at a glance. Juxtaposition in this aspect means use of visual cues such as colors, icons, or annotations to indicate the nature of the differences.
- *Three-Way Comparison:* Juxtaposition in this context involves showing the differences between the common ancestor and the two versions, helping users understand the changes made in each branch.
- *Conflict Resolution:* In collaborative environments, conflicts can arise when multiple users modify the same part of a model independently. Juxtaposition in model comparison tools facilitates conflict resolution by presenting conflicting changes and allowing users to choose which changes to accept, merge, or override.

5.3.2. *superposition*

Superposition, in the context of model comparison, refers to the ability to overlay or combine multiple versions of a model to provide a comprehensive view that includes changes, additions, and deletions from each version. This technique is particularly useful when comparing multiple models or versions simultaneously. Superposition enables a unified representation that incorporates the differences between models, making it easier for users to analyze and understand the changes. Below are some usages of superposition in the model comparison context along with examples of tools that leverage this concept:

- *Unified Model Visualization:* Superposition allows users to view multiple model versions simultaneously in a single, unified visualization. Changes from different versions are overlaid or combined, providing a holistic view.
- *Conflict Resolution:* Superposition aids in identifying and resolving conflicts by presenting overlapping changes from different versions.

5.3.3. *Explicit encoding*

Explicit encoding in the context of model comparison visualization refers to the deliberate and clear representation of differences, additions, and deletions through visual elements. This technique enhances the interpretability of the comparison results by using visual cues to convey specific information. Here are some usages of explicit encoding in the visualization of model comparison:

- *Color-coded Differences:* They uses different colors to represent different types of changes, such as additions, deletions, or modifications. This provides a quick and intuitive way for users to identify and understand the nature of differences.
- *Icons and Symbols:* It means employing specific icons or symbols to represent different types of changes. Icons can be placed next to or within model elements to indicate whether they are added, deleted, or modified.
- *Annotation and Labels:* It conveies adding textual annotations or labels to explicitly describe the nature of changes. This can include information such as the date of modification or the author who made the changes.
- *Line Markings for Relationships:* When comparing models that include relationships between elements, it means using line markings or connectors to indicate changes in

relationships. This makes it clear which connections have been added, deleted, or modified.

5.3.4. *Challenges in the visualization of semantic/domain-specific model comparison*

An important drawback of current mode-based comparison approaches lies in their lack of adequate visualization techniques for domain-specific differencing and merge tasks. Visualizing conflicts in the concrete syntax of different models is a significant challenge that hasn't been addressed for any modeling language (Sharbaf et al, 2022b).

Only a couple of existing methods (e.g., (Wieland et al, 2013; Bartelt and Schindler, 2010)) offer graphical support for conflict resolution, but they fall short of providing a clear overview of the model elements involved in conflict situations (Sharbaf et al, 2022b). Additionally, the visualization approach introduced in (Wieland et al, 2013) only supports manual resolution and focuses solely on fine-grained differences and conflicts. Moreover, the approach in (Bartelt and Schindler, 2010) lacks support for various conflict resolution strategies, and it doesn't provide detailed information on specifying and Visualizing the resolution. This indicates a trend towards graphical domain-specific differencing and merge activities (Sharbaf et al, 2022b). Hence, a promising avenue for future research could involve visually guiding users in describing conflict specifications.

Furthermore, adding a graphical representation of changes in the concrete syntax editor could enhance collaborators' awareness, helping them avoid conflicts during the modeling phase. However, there are only a few approaches that concentrate on conflict visualization (e.g., (Brosch et al, 2012f)) or provide user-friendly graphical editors (e.g., (Mens et al, 2005; Barrett et al, 2011)) for managing conflicts during model merging. Nonetheless, (Brosch et al, 2012f) lacks an editor to specify and visualize conflict and semantic resolution patterns. Moreover, the visualization method in (Mens et al, 2005) relies on cross tables, and the approach in (Barrett et al, 2011) utilizes text-based conflict reports. Thus, the ongoing evolution of graphical and visual solutions for conflict management remains a central challenge in this field (Sharbaf et al, 2022b).

Gleicher (Gleicher, 2018) provides general guidelines for visualizing comparisons. For many different domains, comparing artifacts is a common task and visualizing the comparison often helps. Generally, the visual comparison is displayed using juxtaposition (*e.g.,* as EMFCompare does in Fig. 4.3), superposition, or explicit encoding (like we do in Fig. 4.10).

Brosch et al. (Brosch et al, 2012b) visualize the changes and conflicts in concurrently evolved versions of the same UML model using UML profiles (stereotypes and tagged values). This permits modelers to resolve the conflicts within the UML editor of their choice while using the concrete syntax of the manipulated language. However, this approach is only suitable for UML models whereas we pursue a general approach for arbitrary domain-specific languages.

More similar to our work, the authors in (Schipper et al, 2009) focus on the visualization of diagram differences in the diagrams themselves. The rationale is to help users to understand the modifications immediately. Their proposed visualization includes pop-ups reporting the changes performed in the neighborhood, zooming to changes, collapsing irrelevant parts, and using different colors to represent additions (green), deletions (red), and changes (blue), either in a single diagram or confronting two diagram versions. They have developed a tool that uses EMFCompare for model comparison, as we do. However, their tool only permits visualizing atomic changes, represented by different colors. Furthermore, this work aims to contribute to the field by providing both fine-grained and coarse-grained domain-specific patterns of change. While a detailed presentation of the thesis contributions is forthcoming, this statement serves as an overview of the scope and intentions of our approach. In addition, the visualization associated with each pattern is highly configurable. Other works, such as (Mehra et al, 2005; Ohst et al, 2003), only permit showing changes using different colors or shape styles.

A few works deal with the scalable visualization of differences in the case of large models. To solve this problem, van den Brand et al. (van den Brand et al, 2010) combine a generic visualization framework for metamodel-based languages to show the fine-grained differences, with polymetric views that provide support for zooming and filtering. Wenzel (Wenzel, 2008) also relies on polymetric views to support scalable visualization of differences based on model metrics. Both works are complementary to ours: whereas we provide domain-specificity to the visualization, these other works add a general visualization layer on top.

6. Synthesis

In this section, we present a synthesis that encompasses the noticeable aspects from the literature concerning domain-specific differencing and model merge.

6.1. Model Differencing

Semantic Matching: Some approaches enrich EMFCompare's default matching process by introducing semantic distinctions, aligning with an emphasis on semantic reasoning over differences. Despite maintaining fast time performance, their precision and recall results fall short. Furthermore, they predominantly concentrate on synonymous terms in the changed model elements within the abstract syntax of DSL, as discussed by Addazi et al (2016).

Semantic Lifting: Another approach to improving semantic differencing involves elevating a set of low-level differences to the level of meaningful editing operations. These operations rely on rule-based specifications. However, in this approach, semantic lifting exclusively addresses the abstract syntax of models, neglecting the concrete syntax (Kehrer et al, 2011).

6.2. Semantic Conflict Management

Pattern Languages and Semantic Conflicts: Some existing approaches create a difference model to represent conflicts only in UML class diagrams. The conflict pattern language is another method to specify conflict patterns. However, the languages used to formulate the pattern of conflicts are textual and hard to learn and use by DSL experts (Sharbaf et al, 2020). Addressing this issue could help DSL users specify conflict patterns in a customizable and flexible manner.

Dependency Analysis: Another shortcoming of existing approaches is the disregard or low attention to dependencies among conflicts of different granularity (Langer et al, 2013a). We believe that this concept needs to be addressed by dependency analysis approaches since it could enhance conflict resolution efficiency.

Domain-Specific Conflict Resolution Rules: The shortage of tools for automatic conflict resolution is another aspect that forces DSL users to manually resolve conflicts in most cases. Addressing this issue would allow DSL engineers to expand and tailor rules, providing a comprehensive and customizable solution.

Insightful Manual Conflict Resolution: Although manual conflict resolution is inevitable, existing solutions offer limited options of "keeping left" or "keeping right" (EMF Compare, accessed August 2023; Koegel and Helming, 2010b). Nevertheless, by offering diverse options, especially considering the overlapping between semantic conflicts, versioning tools can propose better manual resolution options. As a result, DSL users can make informed decisions. This could ensure that conflicts are resolved in a way that aligns with the original intention and requirements of the authors.

6.3. A Configurable Visualization

Visualization plays a pivotal role in enhancing understanding, communication, and conceptual clarity in DSMLs. Approaches for domain-specific versioning mainly utilize UML profiles for visualizing changes and formalized constraints using OCL. Therefore, it cannot be useful in DSLs that are limited to UML models and cannot help DSL engineers express complex change, conflict detection, and resolution patterns (Sharbaf and Zamani, 2020). In addition, most of the approaches are limited to atomic changes, lacking support for more coarse-grained difference or conflict patterns.

Moreover, real-world models may have a huge number of model elements or an enormous number of conflicts (de la Vega and Kolovos, 2022). It may create challenges in loading all the difference models, resolving all conflicts at one time, and navigating among conflicts that need to be addressed.

Graphical Support and User-Friendly Editors: Mentioning the gap in visualization techniques for conflict management tasks, Sharbaf et al (2022b) notes the limited graphical support in existing methods. Therefore, there is a need for DSL editors, allowing DSL engineers to specify semantic differences, semantic conflicts, and resolution patterns visually using a comprehensible concrete syntax.

6.4. Versioning Tools

Current commercial versioning tools for models like MagicDraw, Obeo Designer, and JetBrains MPS come short in providing domain-specific aspects in core features of VCS. Addressing their shortages can offer tailored visualizations for any DSL, ensuring flexibility and ease of use for DSL engineers.

6.5. Conclusion

In our examination of the existing literature, we have underscored the imperative of offering domain-specific solutions for model differencing, semantic conflict management, and versioning. Our focus has particularly been on exploring approaches for semantic conflict definition, manual conflict resolution, and configurable visualization. Additionally, our findings emphasize that there is a growing demand for managing the complexities of model evolution and version control not only in the level of abstract syntax but also in the context of concrete syntax, addressing to diverse Domain-Specific Languages (DSLs).

Chapter 3

A Vision on domain-specific differencing and merging of models

This thesis aims to provide a domain-specific approach for differencing, conflict detection, conflict resolution, and merging of domain-specific models. The majority of studies to date offer specific solutions for the model-based differencing and merging problems, mostly focused on abstract syntax without providing a holistic solution. In this thesis, we consider the problem as a whole, and we propose domain-specific differencing and merging solutions for domain-specific models. In the remainder of this chapter, we explain this idea for the main challenges and related solutions explained in Chapter 1.

We start with an overview of the general framework, and then relative to each main challenge, we present the proposed contributions within this framework. The details about each contribution will be given in the subsequent chapters as articles.

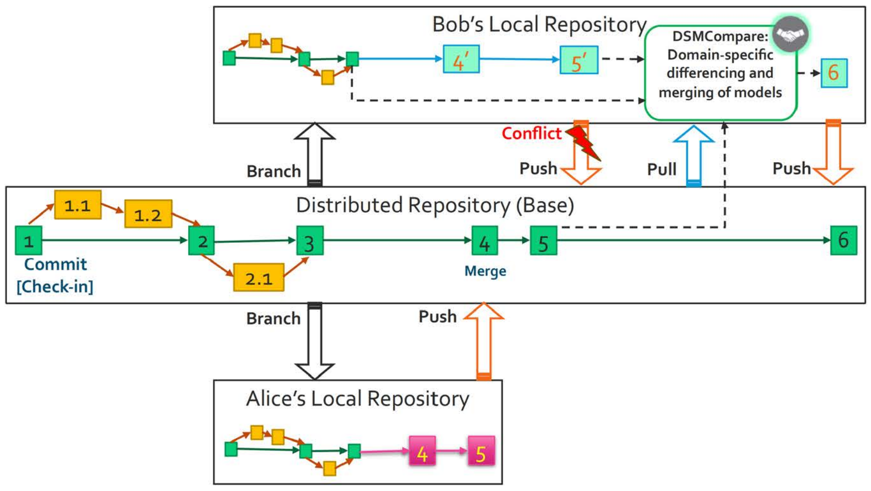


Figure 3.1. 3-way diff and merge in the domain-specific differencing and merging of models

1. Overview

Fig. 3.1 illustrates the primary goal of implementing DSMCompare. DSMCompare aims to assist DSL users in resolving and merging conflicts in a domain-specific manner. As you can see, Alice and Bob create branches to add new features. Alice finishes sooner and merges her changes, but Bob encounters conflicts because the base repository has changed after Alice's successful push. As a result, DSMCompare automatically pulls the latest changes and helps Bob resolve conflicts in a domain-specific way. Finally, it pushes the resolved changes (commit number 6) to the base repository.

The primary challenge in the domain-specific differencing and merging of models is to enhance semantic differencing and visualization. We address this problem by providing the DSL user with a set of *two-way semantic/domain-specific model differences* that highlight the differences between two versions of a model at both the abstract and concrete syntax levels. We also enable the DSL engineer to create the rules for detecting semantic differences in a domain-specific way.

The second main problem is to effectively detect and Visualize the semantic conflicts. We address this problem by providing the DSL user with a set of *three-way semantic/domain-specific model differences and conflicts* that highlight the differences and different types of conflicts between three versions of a model at both the abstract and concrete syntax levels. We also enhance the visualization by providing the layering concepts to improve the user experience.

The third main problem is to empower conflict resolution in domain-specific contexts. We address this problem by the DSL user with *user-friendly conflict resolution mechanism* that generates and highlights the conflict resolutions for each conflict. We also automate merging non-conflicting and equivalent differences, automate resolution for the semantic differences, and provide appropriate visualization and user interface for the DSL experts.

We present in the following, three contributions that target detecting and visualizing semantic differences, semantic conflicts, and conflict resolution and merging.

2. The Design Decisions for Domain-Specific Model Diff and Merge

We outline the essential design decisions for effective domain-specific model differencing and merging.

2.1. Enhancing Semantic Differencing and Visualization

In the first step, we aim to enhance semantic difference extraction to provide more meaningful domain-specific insights. This involves aligning differencing rules with domain characteristics to improve user understanding. The key design decisions for this step are detailed in the following list, and the details are given in Chapter 4, which also illustrates our first article.

Automatic Extension of DSL Meta-model. The proposed solution needs to identify and implement the necessary mechanisms to enable the representation of model differences within the DSL.

Higher-level Representations of Lower-level Differences. This design decision involves defining semantic rules that can analyze and transform low-level differences into more meaningful and understandable representations. It also involves considering possible conflicts that may arise when multiple rules are applied to the same elements of a difference model (Altmanninger et al, 2008a).

Automated Representation of Model Differences. This design decision involves adapting existing Sirius-based editors for model change visualization to automatically represent the differences in a visually intuitive manner.

Prototype Tool Support. DSMCompare should be able to automatically extend the DSL metamodel, apply semantic rules to create higher-level representations of differences and adapt Sirius-based editors for visualization. The tool should also be capable of handling model histories created by third parties and validating the approach on different DSLs and modeling projects (Zadahmad et al, 2022; David et al, 2021).

2.2. Effective Detection and Visualization of Semantic Conflicts

We explore three-way domain-specific differencing and conflict detection, aiming to identify and visualize semantic conflicts for user understanding. Our goal is to enhance conflict resolution and decision-making for future research. The key esign decisions for this step are detailed in the following list, and the details are given in Chapter 5, which also illustrates our second article.

Meta-model Agnostic. The tool accommodates models conforming to any metamodel, ensuring applicability across diverse DSLs.

Concrete Syntax Presentation. Differences are presented within the concrete syntax of the DSL, promoting user familiarity and understanding.

User-Defined Semantics. The tool empowers DSL engineers to define meaningful differences through semantic patterns relevant to the domain.

Fine-Grained Difference Detection. Proficiently detects fine-grained differences using abstract syntax, capturing additions, deletions, modifications, and association rerouting.

Semantic Difference Detection. Identifies semantic differences based on predefined rules, grouping fine-grained changes for meaningful insights.

Equivalent Change Detection. Efficiently identifies identical changes across distinct model versions, reducing redundancy during merging.

Fine-Grained Conflict Detection. Detects conflicts arising from contradicting fine-grained changes, aiding effective conflict resolution.

Semantic Conflict Detection. Extends conflict detection beyond abstract syntax, supporting users in reconciling semantic conflicts.

Explicit Difference Presentation. Explicitly presents differences and conflicts, offering insights through dedicated structures, models, or traces.

Headless API. The tool is accessible interactively or via API, enabling seamless integration with diverse tools and systems.

Three-Way Differencing. Supports three-way comparisons, vital for collaborative scenarios, simplifying conflict reconciliation and merging.

2.3. Empowering Conflict Resolution in Domain-Specific Contexts

In the third step, we empower DSL users to navigate, resolve, and even undo conflict resolutions effectively, improving collaboration and project advancement by addressing domain-specific conflict challenges. The key design decisions for this step are detailed in the following list, and the details are given in Chapter 6, which also illustrates our third article.

DSL-Adapted Conflict Environment. DSL users are provided with an environment that aligns with their familiar DSL syntax, promoting intuitive conflict management. This design decision stems from the realization that users are more efficient in resolving conflicts when presented with a context they are accustomed to. For instance, in our running example of a graphical DSL for designing floor plans, conflict resolution tools should present conflicts within the graphical interface that designers are well-versed in, thereby streamlining the resolution process.

Automated Resolution of Trivial Changes. The tool automatically resolves trivial changes that have been performed in one or both versions. Trivial changes, which are straightforward modifications with negligible impact, are a common occurrence during merging. By automating their resolution, developers are spared the effort of manually addressing these minor alterations, thus expediting the conflict resolution process.

Automatic Resolution of Semantic Conflicts. When viable resolution strategies are available, the tool autonomously resolves semantic conflicts. This feature caters to situations where the nature of the conflict and its possible solutions are well-defined within the DSL's context. For instance, if a DSL for financial modeling encounters a conflict between different interpretations of an interest rate, the tool could apply a predefined mathematical rule to harmonize the conflicting values.

Manual Resolution Support. The tool accommodates manual resolution, granting users the autonomy to tailor conflict resolutions according to specific project problems. Manual intervention is crucial when the nature of the conflict surpasses automated strategies, requiring domain-specific expertise or project-specific considerations for resolution.

Fine-Grained Conflict Navigation. Users are empowered to navigate between conflicts, with the ability to delve into the underlying fine-grained conflicts that contribute to broader, coarse-grained conflicts. This navigation aids users in comprehending the root causes of conflicts, enabling more informed resolution decisions.

Resolution Reversibility. Users can undo previously made resolution decisions for each conflict and modify their choices. This feature ensures flexibility and allows users to iterate on their resolution strategies, refining their approach as the merging process unfolds.

Partial Resolution Saving. For scenarios involving large models or an extensive volume of conflicts, users can save partial resolutions. This functionality enables users to pause the merging process and later resume from where they left off, minimizing disruptions and facilitating effective conflict management.

3. Enhancing semantic differencing and visualization

The first contribution of domain-specific differencing and merging of models is described in Chapter 4, DSMCompare: domain-Specific Model Differencing for Graphical Domain-Specific Languages.

The initial key issue involves improving semantic differencing and visualization. The semantic differences show the intention of the contributor of the branch to make the changes. By understanding the contributor's intention during the merge time, the user responsible for resolving the conflicts can decide in a better way. In addition, semantic differencing can have other usages for a DSL expert. The detection of semantic differences is covered in the research. However, the mechanisms to specify semantic differences and the visualization of related editors to view differences and define rules are not available in a domain-specific manner. To provide a domain-specific approach for detecting and visualizing semantic differences, we sought to generate a metamodel to represent and visualize model differences by extending the original DSL of the domain-specific model. It aims help to fill the comprehension gap required to understand the semantic and fine-grained differences since we provide higher-level representations of lower-level differences in the same concrete syntax. To enable DSL experts to create new rules and maintain the previously defined rules, we automatically generate domain-specific editors allowing DSL engineers to specify patterns for detecting semantic differences. However, the problem is that multiple semantic difference rules may be applicable at the same time, and they might conflict with each other. Therefore, we provide an elaborate graph-based analysis of the rules based on heuristics to obtain a reasonable schedule of the rule application order. In this case, the ordering must be such that it reduces the verbosity of the presented difference, to favor semantic differences over syntactic differences.

4. Effectively detecting and Visualize the semantic conflicts

The second contribution of domain-specific differencing and merging of models, described in Chapter 5, is from two-way to three-way: domain-specific model differencing and conflict detection.

The second primary challenge is to efficiently identify and visualize semantic conflicts. The detection of semantic conflicts is covered in the research. However, the visualization support as well as the domain-specific rule editors is not addressed effectively. On the other hand, the solution we implemented for the first problem was focused on two-way differencing. As a result, we have further enhanced our practice, DSMCompare, to detect semantic differences and conflicts based on a three-way comparison. Through a domain-specific approach, we automatically extended both the abstract syntax and concrete syntax of DSL and generated a three-way difference DSL for abstract syntax and for concrete syntax to represent and visualize both fine-grained and semantic differences and conflicts using Sirius. To accomplish three-way difference and conflict detection, we rely on the preliminary results produced by the three-way merge service that EMFCompare offers. Then we extend the semantic differencing rule DSL and editor from the three-way difference DSL to enable the DSL engineer to specify the semantic difference patterns. The defined rules are transformed to Henshin; first to detect the semantic differences, second to determine an optimized order for the rule application aimed at finding more semantic differences and filtering more fine-grained differences, and third to locate the potential conflict between fine-grained and semantic differences or two semantic differences using Henshin's MultiCDA feature. The result of the last step was a list of potential conflicts, including a minimal model fragment to be checked. We used the generated conflict information in run-time to ensure the performance and accuracy of finding semantic conflicts since we are only checking limited fragments of the difference model for conflicts.

5. Empowering conflict resolution in domain-specific contexts

The third contribution of domain-specific differencing and merging of models, described in Chapter 6, is domain-specific conflict resolution and model merge.

The third primary challenge is to enable conflict resolution in domain-specific contexts. The conflict resolution mechanisms are covered in the research. However, the introduced approaches are mostly focused on abstract syntax, or the approaches to specifying the conflict resolution rules are not given in a way that is easily used by the domain expert. On the other hand, the strategies provided for conflict resolution are very limited and resolve the conflicts in a fine-grained manner and are not focused on resolving all the fine-grained conflicts associated with a semantic conflict at once. As a result, we introduce an approach for domain-specific conflict resolution and model merging based on a three-way comparison.

It automatically generates a domain-specific editor to create conflict resolution rules and enhances the concrete syntax to allow DSL users to visualize the three-way conflict resolutions more effectively. This solution enables DSL users to manage conflicts in an environment familiar to their DSL, navigate between conflicts, manually resolve conflicts that need user intervention, undo previous resolution decisions, and save partial resolutions.

We plan to incorporate a conflict reconciliation mechanism that leverages artificial intelligence techniques to learn implicit user preferences. Additionally, we aim to integrate DSMCompare into domain-specific VCS systems using web-based editors.

Chapter 4

DSMCompare: Domain-Specific Model Differencing for Graphical Domain-Specific Languages [1]

Manouchehr Zadahmad,
Eugene Syriani, Omar Alam, Esther Guerra and Juan de Lara
DIRO, Université de Montréal

This article represents the work I did for the first contribution of my thesis.
Eugene Syriani and Omar Alam are my supervisors, and
Esther Guerra and Juan de Lara are professors at Universidad Autónoma de Madrid
(Spain) who contributed to the writing.

[1]The content of this chapter has been published in Software and Systems Modeling. 2022 Oct 1:1-30. Reference (Zadahmad et al, 2022)

Résumé. Lors du développement d'un projet logiciel, différents développeurs collaborent pour créer et modifier des modèles. Ces modèles évoluent et besoin d'être versionné. Au cours des dernières années, des progrès ont été réalisés dans l'offre d'un support dédié à la gestion des versions de modèles, qui améliore ce qui est pris en charge par les systèmes de contrôle de version basés sur du texte. Cependant, il reste nécessaire de comprendre les différences entre les modèles en termes de sémantique du langage de modélisation et de visualiser les changements en utilisant sa syntaxe concrète. Pour résoudre ces problèmes, nous proposons une approche globale — appelée *DSMCompare* — qui prend en compte à la fois la syntaxe abstraite et concrète d'un langage spécifique à un domaine (DSL) lors de l'expression des différences de modèle, et qui prend en charge la définition de domaines -sémantique spécifique pour des modèles de différences spécifiques. L'approche est basée sur l'extension automatique du DSL pour permettre la représentation des changements et sur l'adaptation automatique de sa syntaxe graphique concrète pour visualiser les différences. De plus, nous permettons la définition de règles de différenciation sémantique pour capturer des modèles de différences récurrents spécifiques à un domaine. Puisque ces règles peuvent entrer en conflit les unes avec les autres, nous introduisons des algorithmes de résolution des conflits et de planification des règles. Pour démontrer l'applicabilité et l'efficacité de notre approche, nous rendons compte d'évaluations basées sur des modèles synthétiques et sur des historiques de versions de modèles développés par des tiers.

Abstract.

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

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

1. Introduction

Model-driven Engineering (MDE) relies on models to conduct all phases of software development. Models can be built using general-purpose modeling languages, e.g. UML, but the use of domain-specific languages (DSLs) is also common (Kelly and Tolvanen, 2008; Schmidt, 2006).

Like other software artifacts involved in a development process, models evolve (Paige et al, 2016) and, therefore, need to be versioned to have a record of their changes (Brosch et al, 2012a). Sometimes, models are persisted as text files (e.g., using the XML metadata interchange format, XMI (OMG, 2023)), which allows using code version control systems on them. However, text-differencing is not adequate for models as it may report irrelevant model differences (e.g., same objects that appear in different file positions). For this reason, the modeling community has proposed specific model versioning systems (Altmanninger et al, 2008a; CDO Model repository, accessed August 2023; Kappel et al, 2006; Koegel and Helming, 2010b) and approaches for model differencing (EMF Compare, accessed August 2023), conflict resolution, and merging (Brosch et al, 2009; Schwägerl et al, 2015).

An important aspect of a versioning system is the ability to visualize matches and differences of the history of a model in a comprehensible manner. However, many approaches, like EMFCompare (EMF Compare, accessed August 2023), represent the differences between two versions of a model using low-level generic traces that may be difficult to understand.

Moreover, these traces typically are at the abstract syntax level, which may further hinder their understanding, since users deal with models using their concrete syntax.

Therefore, we propose to represent traces in a domain-specific way, assign domain-specific semantics to recurring model differences (by defining semantic differencing rules), and visualize those differences at the concrete syntax level. Our approach lifts low level differences between two models to high level differences based on the semantics of the DSL and represents them by reusing the concrete syntax of the DSL. In this paper, we focus on graphical concrete syntaxes realized through the Sirius framework (Sirius, 2023a). Since different semantic rules may conflict with each other, we propose an algorithm to assign priorities to rules, by their automated static analysis. To ensure the practicality of our proposal, we provide automated tool support to minimize the effort of applying the approach to arbitrary graphical DSLs.

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

This article extends our preliminary work (Zadahmad et al, 2019) in several ways. First, we have made several improvements to the semantic differencing rules that encapsulate domain-specific differences. In Section 4.1, we explain how these rules can now express multiple negative application conditions. Also, the new Section 4.3 explains how the semantic differencing rules can be mapped to graph transformation rules. We illustrate our implementation using Henshin. This generalizes our approach, which now can be ported to other modeling frameworks. Second, in (Zadahmad et al, 2019), we assumed that the rules are independent from each other and each rule is applied in isolation. However, in most scenarios, multiple semantic differencing rules may be applied on the same elements of a difference model. Therefore, we have devised an algorithm dedicated to resolving conflicting rules when they are applied in combination, which is presented in Section 5. The algorithm is directed to optimize the verbosity of the domain-specific difference model by suggesting a prioritized list of the rules to the user. Third, we extended the evaluation of our approach in several ways in Section 6. We have refined and extended the research questions to assess the effectiveness of our approach. To answer these, we now include a synthetic experiment, and validate our approach on two modeling projects developed by third-parties: Arduino[2] designer models and evolution of Ecore metamodels.

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

2. Overview and running example

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

2.1. Motivating example

A typical model differencing tool compares two versions of a model based on the performed editing steps (*e.g.,* added class or deleted reference). The result of this comparison is identified by low-level differences between the two versions, which includes at least two sets: *match* and *diff*. The match set establishes a pair-wise correspondence between similar elements in both models. The diff set computes the differences between each pair in the match set. The most popular generic model comparison tools, EMFCompare (EMF Compare, accessed August 2023) for instance, produce three kinds of diffs: ADD, DELETE, and MODIFY.

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

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

There have been approaches that tried to mitigate this issue, *e.g.,* through the semantic lifting of the low-level changes (Kehrer et al, 2011) or by using a metamodel to represent

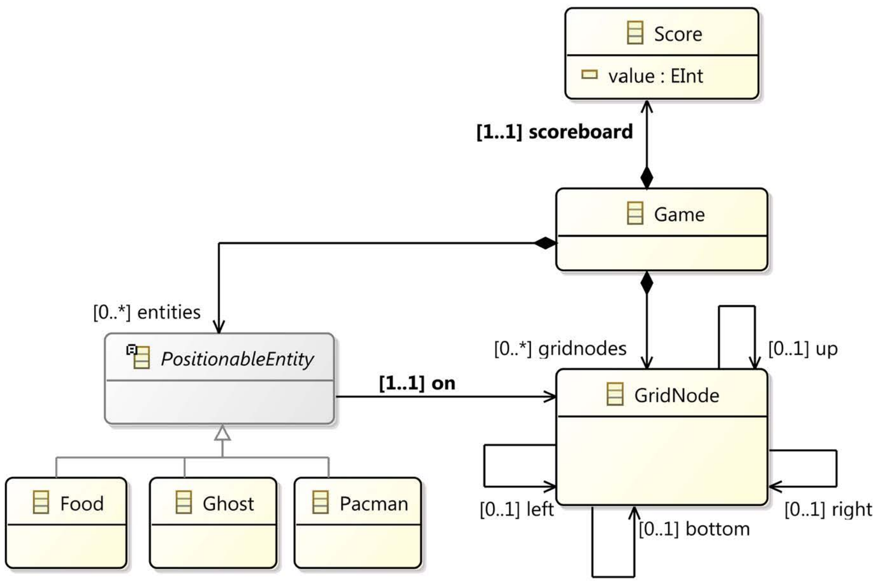


Figure 4.1. Metamodel of the Pacman game DSL

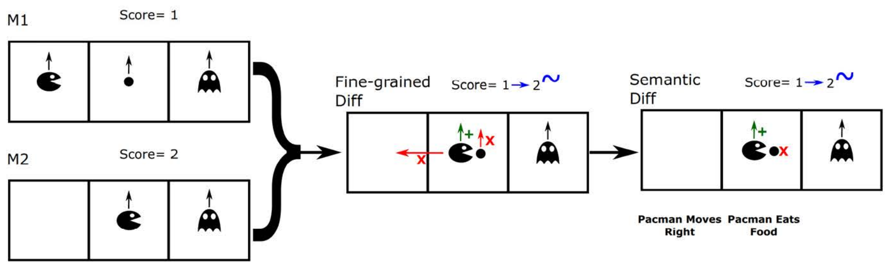


Figure 4.2. Running example using DSMCompare

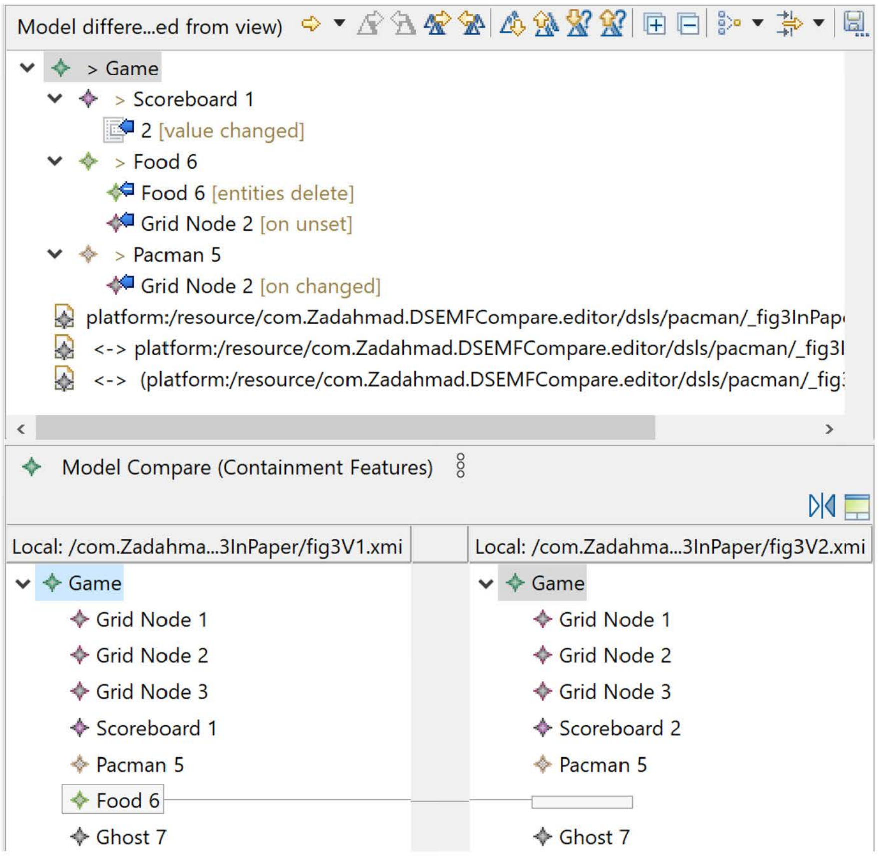

Model differe...ed from view)
> Game
> Scoreboard 1
2 [value changed]
> Food 6
Food 6 [entities delete]
Grid Node 2 [on unset]
> Pacman 5
Grid Node 2 [on changed]
platform:/resource/com.Zadahmad.DSEMFCompare.editor/dsls/pacman/_fig3InPap
<-> platform:/resource/com.Zadahmad.DSEMFCompare.editor/dsls/pacman/_fig3I
<-> (platform:/resource/com.Zadahmad.DSEMFCompare.editor/dsls/pacman/_fig3
Model Compare (Containment Features)
Local: /com.Zadahma...3InPaper/fig3V1.xmi
Local: /com.Zadahma...3InPaper/fig3V2.xmi
Game
Grid Node 1
Grid Node 2
Grid Node 3
Scoreboard 1
Pacman 5
Food 6
Ghost 7
Game
Grid Node 1
Grid Node 2
Grid Node 3
Scoreboard 2
Pacman 5
Ghost 7

Figure 4.3. Representation of difference model in EMFCompare for the Pacman game DSL

model differences (Cicchetti et al, 2007). However, these approaches do not provide a comprehensive framework for handling domain-specific model differences. In particular, the existing approaches mostly focus on expressing model differences at the abstract syntax level and do not show differences at the concrete syntax level (*i.e.,* the graphical notation of a DSL). Furthermore, the existing approaches do not take domain-specific model semantics[3] into consideration during the comparison process.

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

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

We provide a modeling environment to define game configurations, based on (Syriani and Vangheluwe, 2013). Fig. 5.2 shows the metamodel of this game. Fig. 4.2 sketches what DSMCompare outputs given two versions (M1 followed by M2) of a Pacman game configuration. The black arrows pointing up over Pacman, food, and the ghost are the associations representing their position on a grid node. Comparing M1 and M2, we can easily conclude that Pacman has moved right to the middle grid node and ate the food on it. The score value is incremented accordingly. DSMCompare produces a domain-specific difference model $Diff_{12}$ in two steps. First, in the middle of Fig. 4.2, $Diff_{12}$ contains all the fine-grained diffs. The green arrow with a '+' denotes that an association is added to a grid node, the red arrow with an 'x' denotes a deleted association, and the blue arrow with a $\sim$ (on the scoreboard) denotes an attribute value change. Then, DSMCompare applies the provided semantic differencing rules on $Diff_{12}$. In this case, two rules can be applied: *Pacman Eats Food* and *Pacman Moves Right*. For example, the former rule checks that Pacman is on a grid node that also has food on it which gets deleted, and the scoreboard value is incremented. The final difference model $Diff_{12}$ is depicted at the right of Fig. 4.2 (labelled as *semantic diff*).

In contrast, using EMFCompare for comparison results in a list of low-level changes as presented in Fig. 4.3. The DSL user needs additional analytical effort to understand these changes to infer the difference in a meaningful way. For example, the user needs to understand that (on the top panel of Fig. 4.3) "on changed" means that Pacman has moved to a different grid node (because the reference "on" has changed), and needs to inspect the lower juxtaposed panels to understand that food has disappeared. However, as the "on" reference is not shown on the tree editor, it becomes difficult to realize that this is because Pacman ate the food.

2.2. Overview of DSMCompare

Fig. 4.4 gives an overview of DSMCompare. The approach is useful for two types of users: DSL engineers (who build the DSL) and DSL users (who create models using the DSL).

To define the DSL, the engineer creates a metamodel *MM* for the abstract syntax, and a model *CS* of the concrete syntax. In DSMCompare, we reuse both components to define the domain-specific model differences for that DSL and show any domain-specific diff

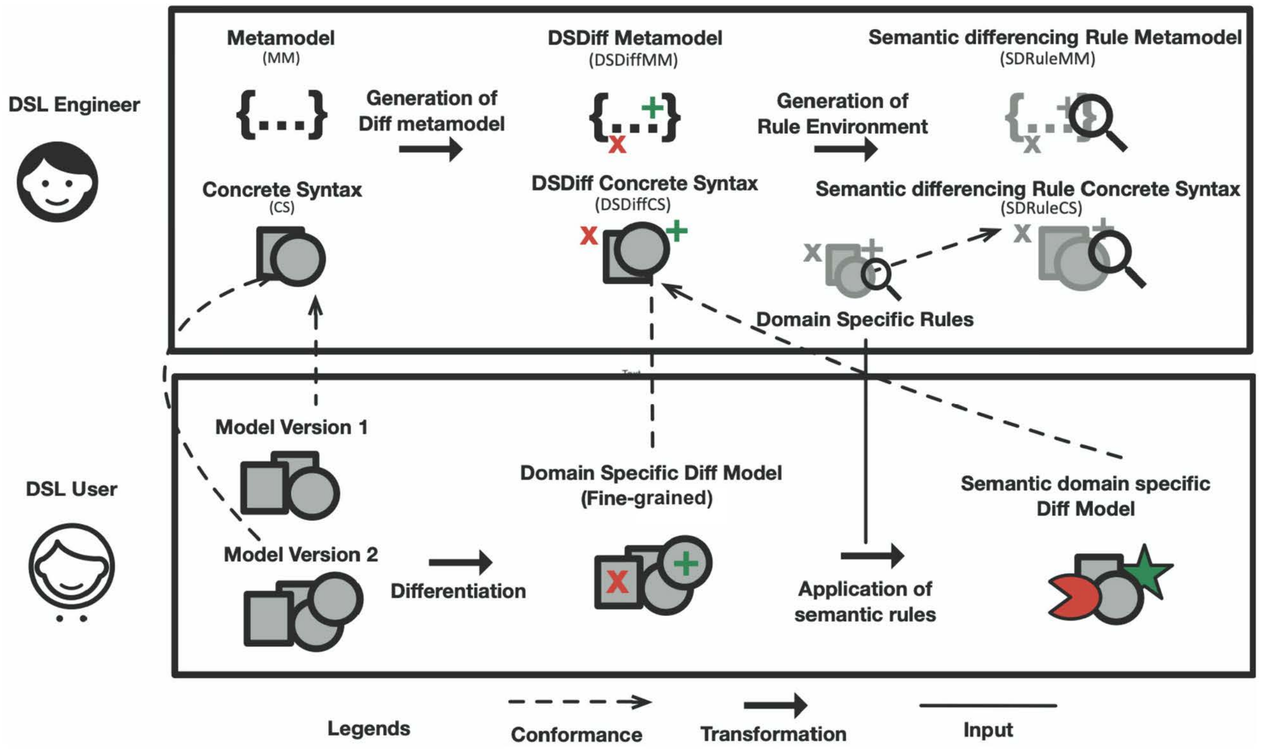


Figure 4.4. Overview of the approach

Diff$_{12}$ between two versions of a model *M1* and *M2*. Concretely, the approach produces a domain-specific diff metamodel *DSDiffMM* and concrete syntax model *DSDiffCS*, as shown in Fig. 4.4. *DSDiffMM* extends the language metamodel to define domain-specific diffs, such as adding/removing a model element. *DSDiffCS* shows the corresponding concrete syntax elements: graphical elements that could be added, removed, or updated.

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

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

DSMCompare produces a traditional diff of the two model versions by reusing a difference tool such as EMFCompare. This result is processed to generate Diff_{12}, that conforms to *DSDiffMM*, and is represented according to *DSDiffCS*. At this point, Diff_{12} contains the fine-grained differences in the concrete syntax of the DSL. With a library of rules predefined by the DSL engineer, the approach executes the applicable rules on Diff_{12} to produce a semantically lifted difference model.

3. Fine-grained differencing

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

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

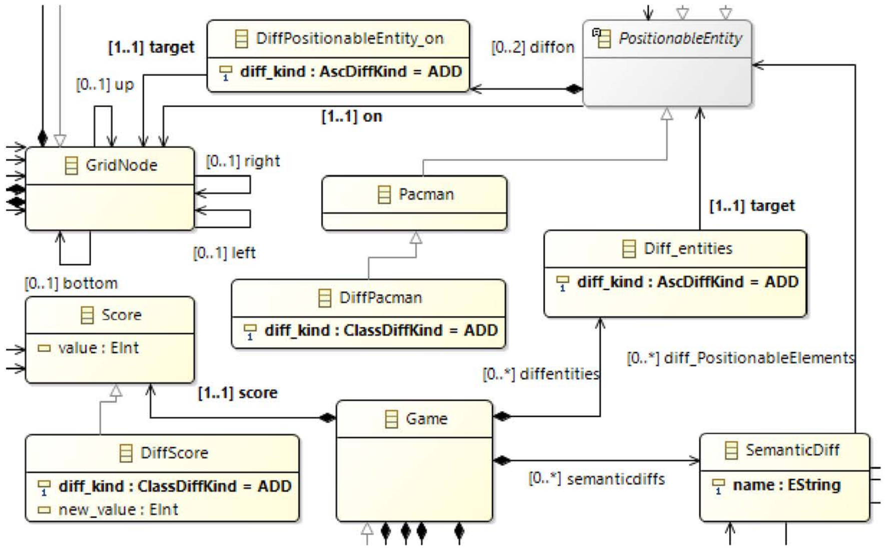


Figure 4.5. Excerpt of the generated difference metamodel

3.1. Domain-specific difference metamodel

To represent model differences in a domain-specific way, the metamodel of model differences should remain faithful to the original metamodel *MM*. Therefore, we create a new metamodel *DSDiffMM* for domain-specific differencing (see Fig. 4.5) based on *MM* (see Fig. 5.2).

Algorithm 1 outlines the transformation from *MM* to *DSDiffMM*. It starts by cloning *MM* to ensure that *DSDiffMM* comprises all the structural features of the DSL. In Fig. 4.5, *DSDiffMM* includes all classes and associations that the *MM* metamodel possesses. The remaining steps extend the metamodel as follows. We create two enumerations that will be used to annotate each class and association with the kind of difference. To represent a difference in an object of a class, like `Score`, we create a subclass with an additional attribute `diff_kind` that states whether the object has been added, deleted, or that at least one of its attributes has been modified. In the subclass we also add, for each attribute in the class, a new attribute of the same type that will hold the new value. For example, the subclass of `Score` has an attribute `new_value`. This is particularly useful when auditing changes in different versions of a same model.

Note that this procedure does not transform the class inheritance hierarchies. If *MM* has a class `A` and a class `B` that inherits from `A`, then, in *DSDiffMM*, `DiffA` inherits from `A` and `DiffB` inherits from `B`, but there is no inheritance between `DiffA` and `DiffB`. We argue that this

Algorithm 1 Transformation from MM to DSDiffMM

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

decision is to allow implementing our solution in frameworks where multiple inheritance is not supported. Therefore, on line 11 of Algorithm 1, `C.getAllUniqueAttributes()` retrieves all attributes of `C` and those inherited from its super classes transitively. Furthermore, abstract classes have no corresponding *Diff* class since they cannot be instantiated in the compared models.

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

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

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

3.2. Visualization of domain-specific differences

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

Sirius (Sirius, 2023a) is one of the most popular frameworks to generate graphical modeling environments and to manipulate models graphically in the Eclipse ecosystem. Although our approach is applicable to other graphical language workbenches, such as GMF (GMF, last accessed 2023), MetaEdit+ (Kelly et al, 1996) and AToMPM (Syriani et al, 2013), our description is based on Sirius because its wide use nowadays, and because it offers a model-based approach for concrete syntax definition.

In Sirius, the main component of the concrete syntax definition is a viewpoint specification model (`odesign`). It defines a mapping of graphical representations to elements of *MM*. For example, to render the visualization of the `Pacman` class, we define a `NodeMapping` that refers to an icon in an image file. The `NodeMapping` can be a combination of text, icons, shapes and style customizations, such as color and size. Similarly, associations are rendered by an `EdgeMapping`. As for compositions, the target class is rendered by a `BorderedNodeMapping` within the `NodeMapping` of the source class. Constraints expressed in the Acceleo Query Language (AQL), a variant of the Object Constraint Language (OCL) (Object Management Group (OMG), 2023), can filter visualizations depending on a condition. Finally, it is possible to define a palette of buttons to instantiate *MM* classes and associations by customizing the `ToolSection`.

We generate *DSDiffCS* by means of an outplace transformation[4] that takes as input *CS* and outputs *DSDiffCS*. The overall logic of the transformation is to copy each component of *CS* onto *DSDiffCS* and create the representation of each `Diff_` class by extending the representation of its corresponding *MM* class. This maximizes the reuse of *CS* to represent the difference model intuitively for the DSL user. For each `NodeMapping`, *e.g.,* `PacmanNode`, we create three new ones for each difference kind: `DiffPacmanNodeADD`, `DiffPacmanNodeDELETE`, `DiffPacmanNodeMODIFY`. By default, the add node is the same as the original node annotated with a green '+' sign, the delete with a red 'x', and modify with a blue '$\sim$'. The latter indicates that at least one of the attribute values has changed. For example, the `ScoreNode` is a rectangle with the value of its `value` attribute displayed inside. We change the text

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

displayed in DiffScoreNodeMODIFY by showing the value concatenated with an arrow '->', followed by the new_value. One particularity of the mapping in Sirius is that if DiffPacman inherits from Pacman in *DSDiffMM*, Sirius displays the representation of the former for the latter. Therefore we need to add an AQL condition in DiffPacmanNodeADD to force it to represent DiffPacman instances only and not its super classes.

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

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

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

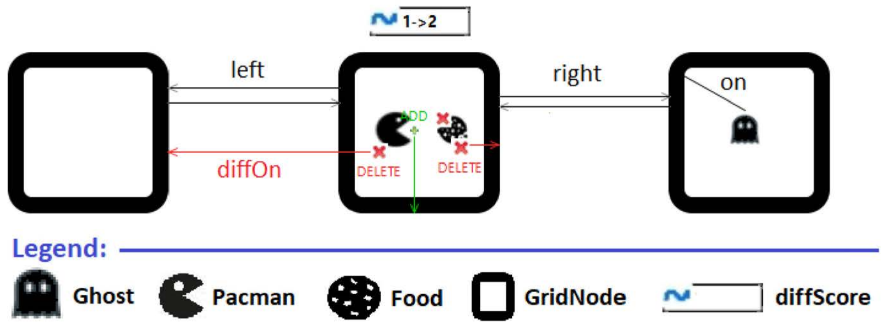


Figure 4.6. Fine-grained difference model Diff_{12} of *M1* and *M2*

Defining *DSDiffMM* along with *DSDiffCS* as a domain-specific difference language using frameworks such as Sirius, allows the DSL engineer to generate a domain-specific model environment to represent difference models $Diff_{12}$. These can be inspected and manipulated like any other model (*M1* and *M2*) in an environment familiar to the DSL user. Fig. 4.6 illustrates the $Diff_{12}$ model for the running example (cf. Fig. 4.2), presented in its concrete syntax as output by DSMCompare.

3.3. Fine-grained domain-specific model comparison

Given two models *M1* and *M2* of a DSL, we want to output a single model $Diff_{12}$ depicting the changes from *M1* to *M2*, as an instance of *DSDiffMM.* Note that the two models are provided with their abstract and concrete syntax representations. Most current model comparison approaches detect changes at the abstract syntax level only. For instance, (Lin et al, 2007) dynamically computes an identifier for each model element based on their properties (*e.g.,* type and attribute values). Alternatively, metamodel-agnostic approaches, like (Brun and Pierantonio, 2008; Cicchetti et al, 2007), compute the structural and attribute value similarities between *M1* and *M2.* These tools produce a generic difference model that lists the changes between the two models. We chose to reuse these difference algorithms and then process the result to produce $Diff_{12}$. In our implementation, we rely on the change list output by EMFCompare.

To produce the $Diff_{12}$ model, we first clone *M1* since the differences will be expressed in terms of *M1.* We assume that the result from a difference algorithm outputs a list Δ_C of differences for classes, and another one Δ_A for associations, such as the case in EMFCompare. We denote an element $E' \in \Delta_C$ using primed uppercase letters. This way, if E' is a deletion or a modification, we identify E to be the corresponding element in *M1.* For example, in Fig. 4.6, E' can be the score object with its value modified from 1 to 2. We replace E', the score object, in *M1* by an instance of the `DiffScore` class as per Algorithm 1. This new object will hold all original attribute values, so `score=1`, and all new attribute values, so `new_score=2`. If E' is an addition, we create an instance of the Diff class corresponding to E' and set all its new attribute values. Finally, we mark the new Diff element with its `ClassDiffKind`.

An association $A' \in \Delta_A$ is treated a bit differently. If A' is a deletion, we remove the link A in *M1* corresponding to A' and create an instance of the Diff class corresponding to it. For example, in Fig. 4.6, the `on` link from the Pacman to the first grid node is removed and an instance of `DiffPositionableElement_on` is created. In case A' is an addition, only the creation of the Diff class is needed. We then connect the Diff instance to the source and target elements of A. Finally, we mark it with its `AscDiffKind`.

Our approach does not require additional manual effort to produce the concrete syntax of Diff_{12}. Since Diff_{12} is an instance of *DSDiffMM*, then *DSDiffCS* is applied automatically on Diff_{12} to represent it visually, as shown in Fig. 4.6.

4. Domain-specific semantic differencing

In this section we introduce the approach to create semantic diff rules. This involves synthesizing a metamodel *SDRuleMM* out of *DSDiffMM*, as we explain in Section 4.1. Then, in Section 4.2 we outline how to generate a graphical environment for the DSL engineer that supports the creation of semantic differencing rules, based on *SDRuleMM*. Finally, Section 4.3 provides a semantics for domain-specific diff rules in terms of graph transformation rules (Ehrig et al, 2006).

4.1. Rules for domain-specific differences

As explained in Section 2, we automatically derive an environment for specifying semantic differencing rules. This enables the DSL engineer to define higher-level changes specifically tailored for the domain. A rule needs to detect a pattern of fine-grained differences and replace it with a `SemanticDiff` class that was created in Algorithm 1. Our semantic differencing rules act similarly to inplace model transformation rules (Ehrig et al, 2006) with a precondition and a postcondition component. Algorithm 2 outlines the procedure to produce *SDRuleMM* from *DSDiffMM* and Fig. 4.7 shows the result. It is inspired by (Kühne et al, 2009) where the authors produce domain-specific model transformation rule patterns from a DSL.

Like Algorithm 1, this procedure starts by reusing all the elements of *DSDiffMM*, adapting them to the new needs. Every class and association is prefixed with `Pattern_`, except the `SemanticDiff` class. All attributes from *DSDiffMM* except `diff_kind` are removed, since

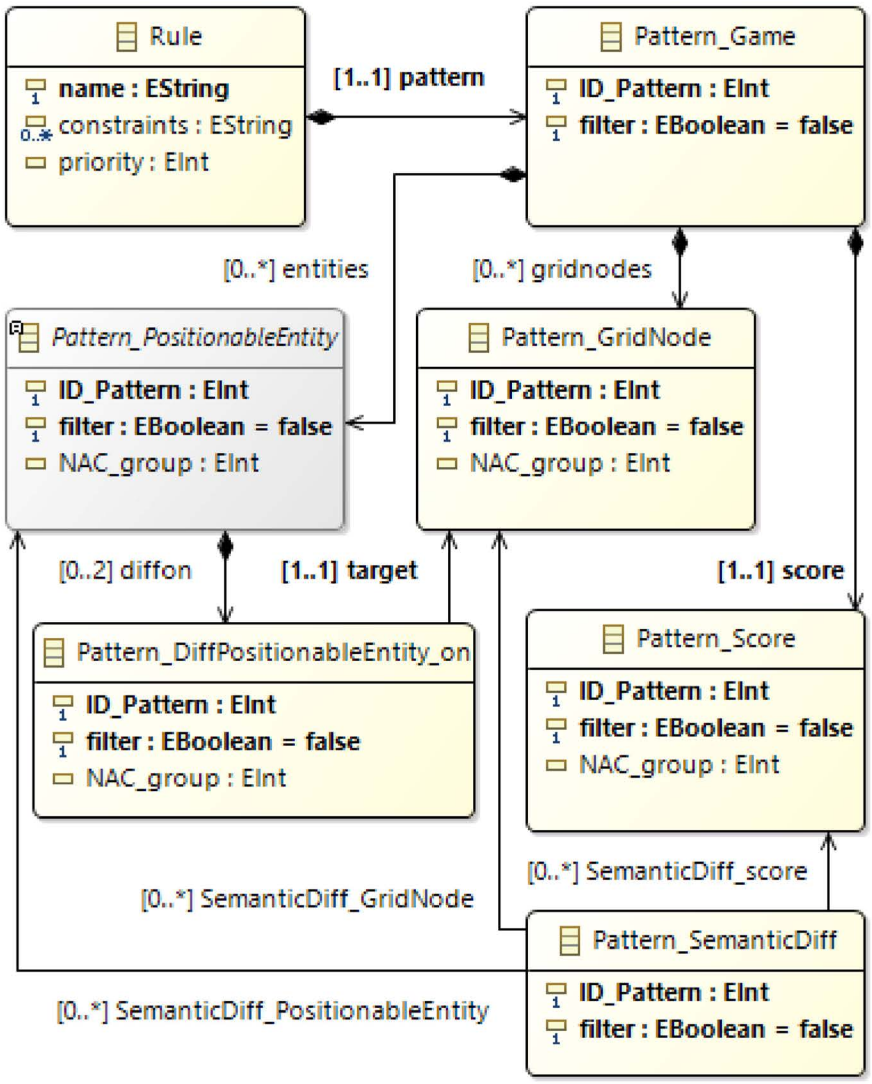


Figure 4.7. Excerpt of the semantic differencing rule metamodel *PacmanRuleMM*
.

they do not contribute to the rule. However, the connectivity of the associations remains as in *DSDiffMM*. This simplifies the detection of patterns in the difference model *Diff*$_{12}$.

We add two attributes to all pattern classes. First, a unique identifier distinguishes instances of the same classes to facilitate writing constraints. Then, a *filter* attribute is used to signify that the element in *Diff*$_{12}$ should be removed when applying the rule. It is helpful to remove fine-grained differences when a domain-specific difference is more meaningful. Furthermore, the rule may contain negative application conditions (NACs) to forbid the presence of elements (Ehrig et al, 2006). We add a `NAC_group` attribute to all classes prefixed with `Pattern_`. Similar to some transformation languages (Arendt et al, 2010), one or more

Algorithm 2 Transformation from `DSDiffMM` to `SDRuleMM`

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

rule elements set with the same `NAC_group` value constitute a NAC. Multiple values of this attribute are used to represent several NACs in the rule, none of which can be matched for the rule to be applicable.

Finally, lines 13–16 of the algorithm add a new `Rule` class as the new root of the metamodel. This enables the transformation engine to navigate easily through the elements of the rule. In addition, the `Rule` class allows specifying a list of constraints over attribute values. In practice, constraints are written in Java and executed dynamically using BeanShell^5, an embedded interpreter to run Java scripts. Within constraints, pattern objects (elements of the rule) can be accessed through the `Item` keyword, using their identifier and the desired attribute name in the form of `Item(ID,[ATTR_NAME])`. Fig. 4.8 shows an example semantic rule called `Eat` (in concrete syntax) with a constraint. This constraint states that the new value of the score should be greater than the original value for the rule to be applicable.

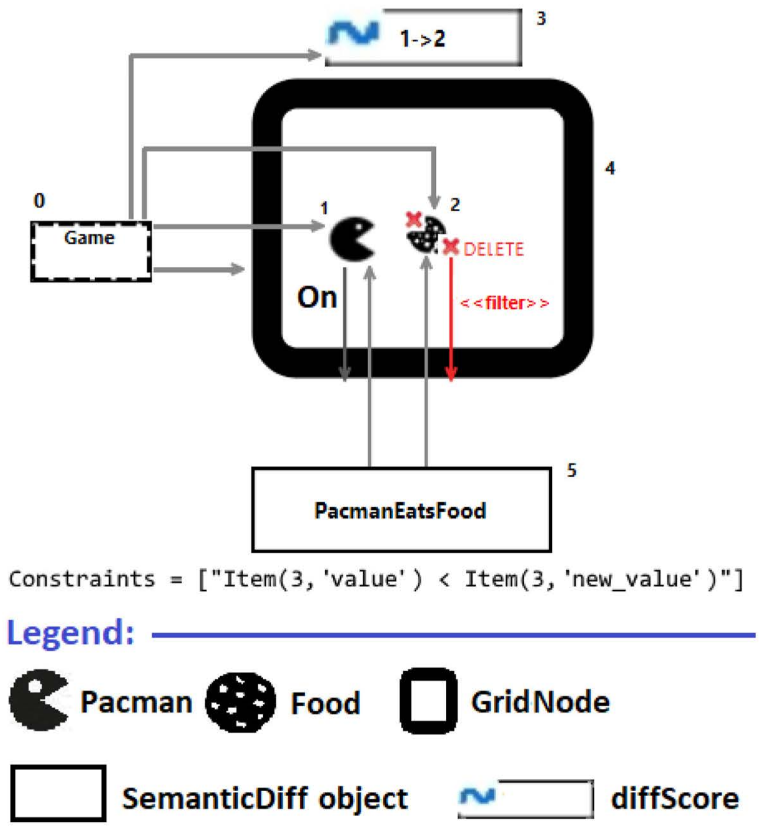


Figure 4.8. The semantic differencing rule `Eat`, abstracting fine-grained differences to depict that Pacman has eaten food

4.2. Automatic generation of a graphical environment for semantic diff rules

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

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

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

Fig. 4.8 illustrates a rule in the generated domain-specific environment. The rule describes that a *Pacman eat food* change occurs when Pacman is on a grid node, a food is deleted from the same node, and the score is incremented. To reduce the amount of fine-grained differences reported to the DSL user, the rule also filters the `on` association from the food to the grid node.

4.3. Executing the semantic diff rules

As outlined in Fig. 4.4, we apply the semantic diff rules to enhance the fine-grained difference model $Diff_{12}$ with semantic differences, and possibly remove fine-grained differences. Given the difference model $Diff_{12}$ produced as described in Section 3.3, we apply the rules on $Diff_{12}$ as an inplace model transformation. For this purpose, we express the semantics of our semantic diff rules as graph transformation rules. In particular, we use Henshin (Arendt et al, 2010) as the target transformation engine. Henshin is an inplace model transformation language implementing graph transformations for the Eclipse Modeling Framework. Therefore, we opted to transform each SDRule into a semantically equivalent Henshin rule, which can then be applied on $Diff_{12}$. In practice, we implemented this higher-order transformation using an Xtend-based code generator. This takes a set of semantic differencing rules and produces a set of Henshin rules. We chose a code generator approach since Henshin rules can be specified in a textual notation (Strüber et al, 2017).

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

For example, the `Eat` rule in Fig. 4.8 looks for a `Pacman` object and a deleted `DiffFood` on the same grid node. It also requires that the new value of `DiffScore` has increased. Then, it creates the `SemanticDiff` object named `PacmanEatsFood` and hides the deleted `DiffPositionableElement_on` link associated with `DiffFood`. Fig. 4.9 shows how this rule is encoded in Henshin. A Henshin rule HRule consists of nodes, edges, and conditions.

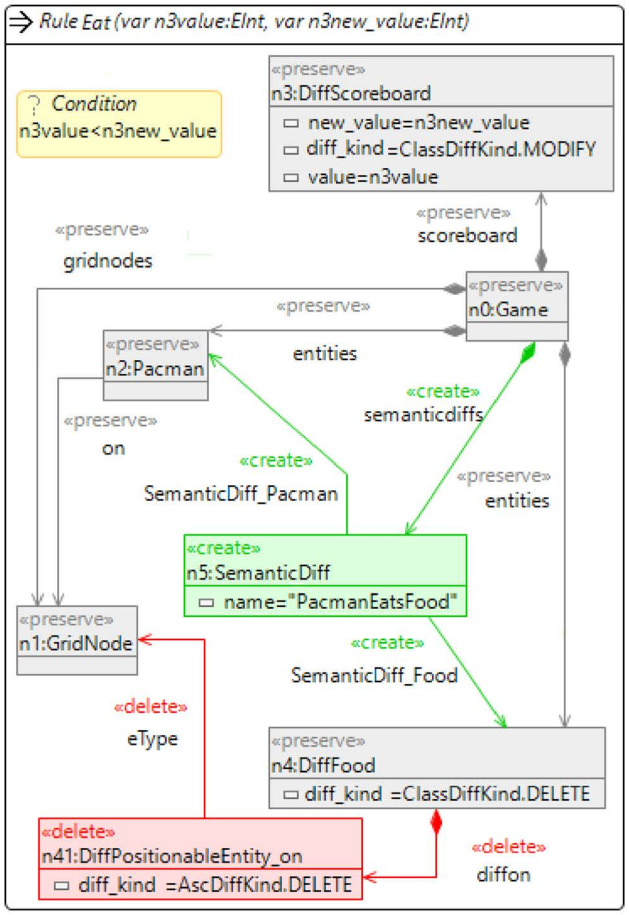


Figure 4.9. Rule `Eat` transformed into Henshin

Nodes and edges can be assigned actions (preserve, create, delete, forbid) and are typed by a metamodel class or association respectively. Nodes can have attribute values.

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

(1) Create an HRule with the same name as the SDRule (line 2 of Algorithm 3).

(2) Create a condition in HRule for every condition in SDRule. If a condition uses an attribute, add a parameter to the rule, then assign the parameter to the corresponding attribute and use the parameter instead of the attribute in the condition (lines 4–8).

(3) Create a node with action «preserve» in HRule for every pattern object with no filter and no `NAC_group` set in SDRule (lines 11–13).

(4) Create a node with action «delete» in HRule for every pattern object with filter set to true in SDRule (lines 14–15).

(5) Create a node with action «forbid» in HRule for every pattern object with a `NAC_group` set in SDRule. Set the forbid identifier to the value of the NAC group (lines 16–18).

(6) Create a node with action «create» in HRule for every `SemanticDiff` object in SDRule (lines 19–20).

(7) If a pattern object has a value for its attributes like `diff_kind` set in SDRule, create the same attribute with the same value in the corresponding Henshin node (lines 22–26).

(8) Create an edge in HRule for each association in SDRule. The type of the edge should correspond to the one of the association (lines 28–41) as follows. All edges adjacent to a node of type `SemanticDiff` have the action «create» (lines 34–35). All edges adjacent to a node with action «delete» or «forbid» have also the action «delete» or «forbid» respectively (lines 36–39). Otherwise, the edge action is set to «preserve» (lines 40–41).

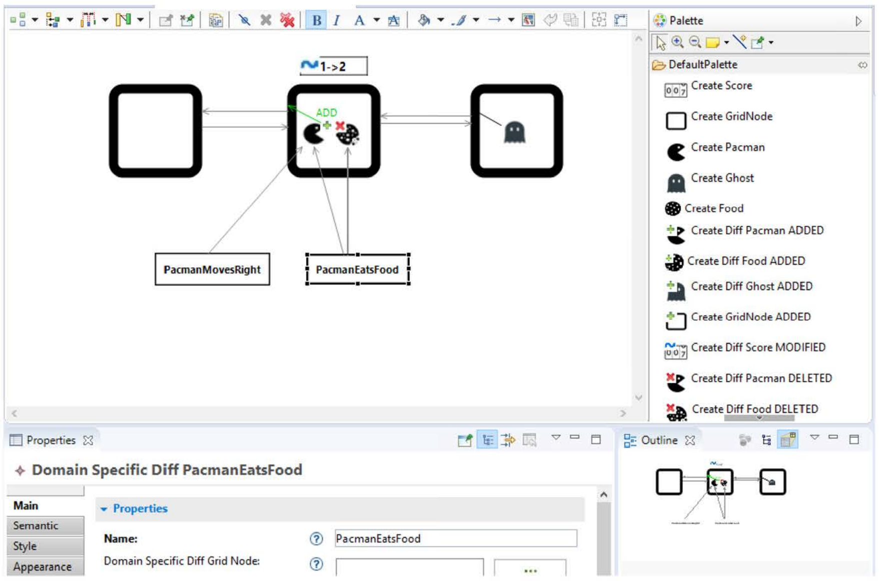
Thanks to the transformation to Henshin, our rules support matching a subclass of a pattern class (Biermann et al, 2012): in *DSDiffMM*, the `DiffScore` class inherits from the `Score` class. Furthermore, abstract classes from *MM*, like `PositionableElement`, can be used when specifying patterns, which can be useful to define fewer rules (de Lara et al, 2007).

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

Fig. 4.10 illustrates the final difference model provided by our approach. It shows the application of two rules, identifying that Pacman has moved right and eaten food. Altogether,

Algorithm 3 Transformation from `SDRule` to `HRule`

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



Domain Specific Diff PacmanEatsFood

Properties

Name: PacmanEatsFood

Domain Specific Diff Grid Node:

Figure 4.10. The domain-specific difference of two models in the generated editor after applying two rules

compared to Fig. 4.3, the DSL user can inspect the domain-specific changes in an editor that resembles the one she used to manipulate the original models *M1* and *M2*.

5. Conflicting rule application

A rule may have more than one match in Diff_{12}. However, care should be taken since applying a rule may remove filtered elements. In general, there is normally more than one semantic differencing rule specified for a DSL and different rules may have overlapping matches. In Section 4.3, the control flow of the transformation assumed the rules are sequentially independent (Ehrig et al, 2006). However, if a rule filters an element that is required in the precondition of another rule, the latter will not find a match. One solution to avoid conflicts between rules is to use NACs. For example, we can prevent the application of a rule if another rule has been applied before. This can be achieved by adding a `SemanticDiff` object in the former rule as a NAC (see Section 4.1). However, this solution is limited because it alters the semantics of the rule, may prevent non-conflicting rules from applying, and requires modifying the semantic rule manually. Therefore, we propose a general solution that reduces conflicts between rules as much as possible.

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

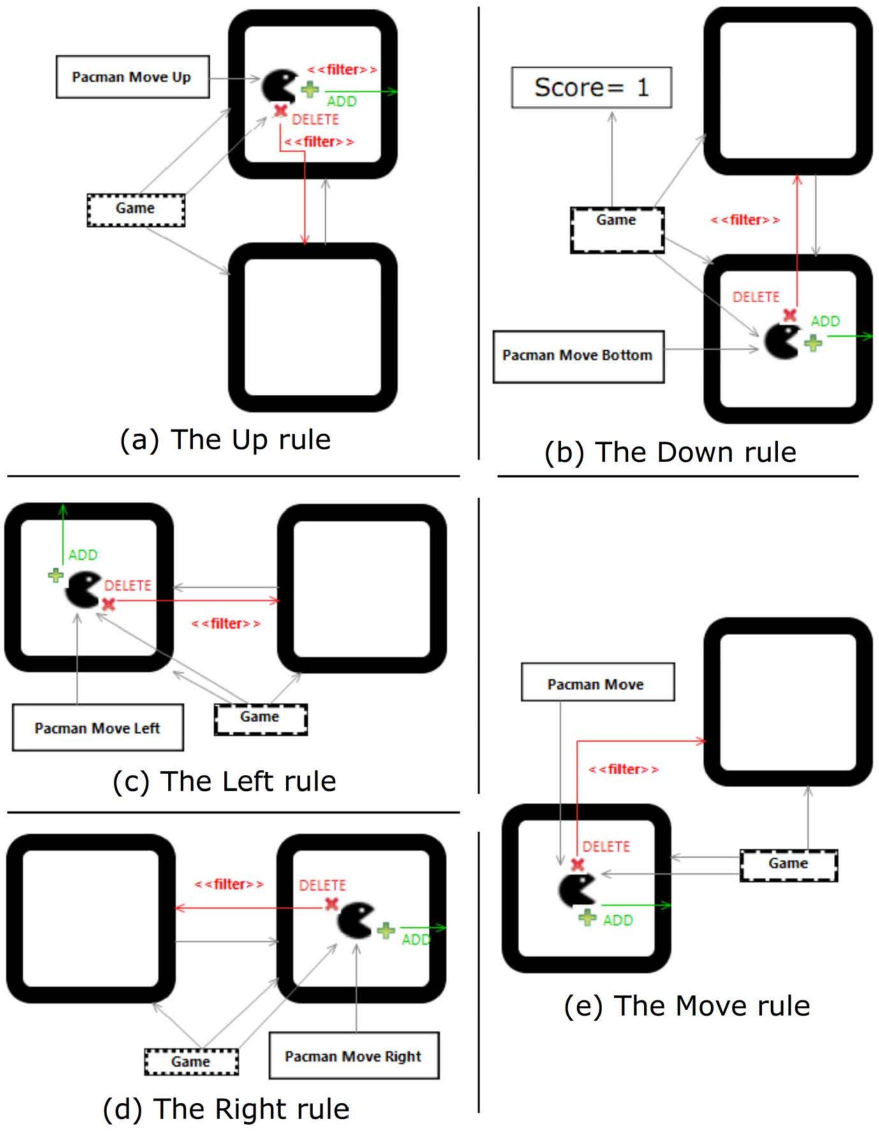


Figure 4.11. Semantic differencing rules for Pacman movement

5.1. Conflicting rules example

Assume the engineer of the Pacman game DSL has defined the semantic differencing rules for the four cardinal movements of Pacman as shown in Fig. 4.11 (a)–(d). Note that we have slightly altered the rules for illustrative purposes. After a while, some DSL users report that DSMCompare fails to detect other kinds of movements, such as diagonally or further than one grid node away. Thus, the DSL engineer creates a new rule called `Move` as depicted in Fig. 4.11 (e). This semantic differencing rule correctly detects any change in Pacman movements. However, later, a DSL user discovers that, for some difference models, DSMCompare reports `Move` instead of the more precise `Right`. This new problem arises because the two rules conflict with each other (when `Move` is applied before `Right`): the former rule filters the old `diff_on` relation of Pacman which is required to apply the latter rule. Another situation occurs when `Move` and `Up` are both applicable, but the former is applied. In this case, the resulting $Diff_{12}$ model will contain more fine-grained differences than if the latter was applied (because `Move` filters one association, while `Up` filters two), thus encumbering the DSL user with unnecessary differences reported. This problem is further aggravated when rules have many occurrences in $Diff_{12}$. This example illustrates that, when a number of rules are in conflict, the DSL engineer should prioritize those that are more precise, remove more fine-grained differences, and create more domain-specific differences.

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

5.2. Formalization of the problem

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

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

5.2.1. *Conflict graph*

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

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

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

The vertices of the conflict graph in Fig. 4.12 show the properties of each rule of the Pacman game presented in Fig. 4.11. An edge $(v_1, v_2) \in E$ represents a conflict that occurs if we apply the rule corresponding to v_1 before the rule corresponding to v_2. Since we assume that a rule is applied on all matches exhaustively before applying another one, edges cannot be reflexive.

Edges are weighted by function $conf$, which gives the number of conflicts that arise when applying the rule of the source vertex before the rule of the target vertex. Following the theory of graph transformation with NACs (Lambers et al, 2006), we consider two kinds of conflicts for rules:

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

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

5.2.2. *Conflict detection*

To compute the edges of the conflict graph and their weight, we perform a conflict analysis of the rules. Henshin offers a multi-granular conflict and dependency analysis tool (MultiCDA), a generalization of critical pair analysis (CPA) (Lambers et al, 2018). Conflicts need to be detected only once by the DSL engineer, thus the computation time of conflicts is not an issue for our problem. Nevertheless, MultiCDA is significantly faster than CPA (Lambers et al, 2018). Given a set of Henshin rules, MultiCDA outputs three levels of conflict granularity. To assign the $conf$ weight to each edge of the conflict graph, we rely on the fine-granularity level that MultiCDA reports. It outputs a positive integer for each pair of rules representing the number of all model fragments whose presence leads to a conflict. MultiCDA presents the conflict results as a matrix. This serves as the adjacency matrix of our conflict graph (note that we assign 0 to the main diagonal since edges are irreflexive).

Applying conflict detection with MultiCDA on the Pacman game semantic differencing rules in Fig. 4.11 results in the edges of the graph in Fig. 4.12. The `Eat` rule has no conflicting model fragment with any other rules, thus it is disconnected. The edges outgoing from `Move` indicate that if we apply this rule before any of the other movement rules, there are six model fragments that lead to conflicts. In contrast, applying any of the cardinal

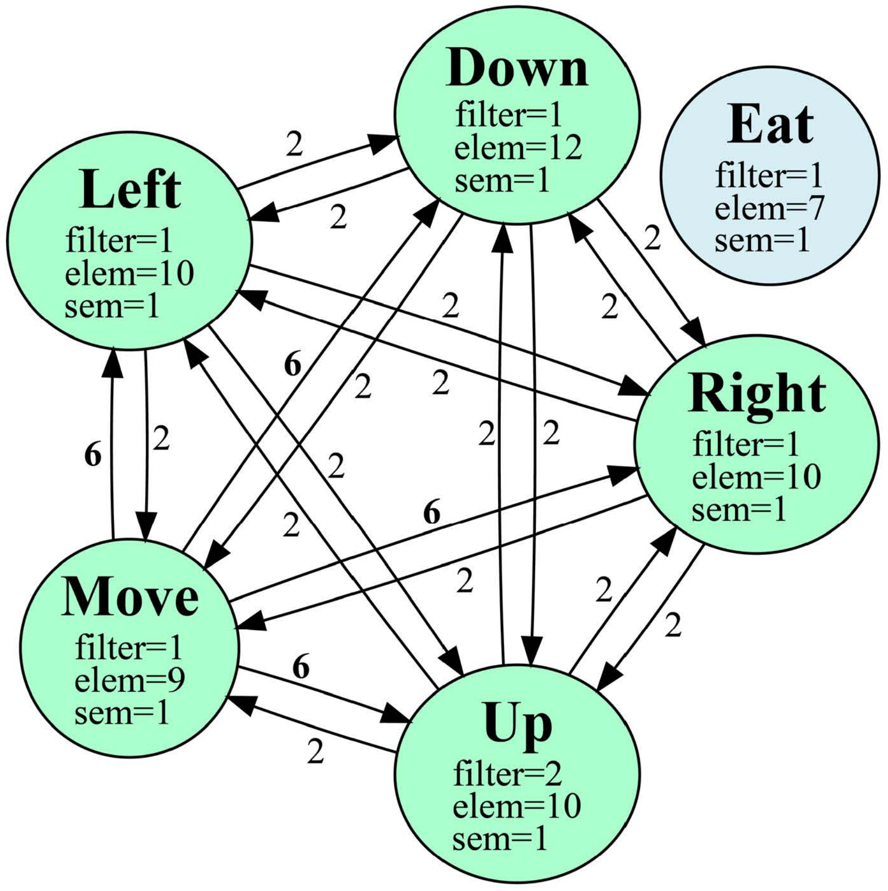


Figure 4.12. The conflict graph for the rules in Fig. 4.11

movement rules before any other movement rule causes conflicts only for two model fragments. For example, one of them is: `[Pacman]-(diffon)-[DiffPositionableEntity_on]-(eType)-[GridNode]`. Applying the `Move` rule on this model fragment will remove the three central elements, whereas all the other movement rules require this fragment to be applicable.

5.3. Rule priority ordering

To illustrate how to solve the rule priority ordering, consider the conflict graph in Fig. 4.13. It represents the conflicts between four semantic differencing rules A,B,C, and D encoded by vertices with the same name. Intuitively, a solution to the problem is to sort the vertices of the conflict graph topologically. However, recall that the edge (B,D) means that when B is applied on a model fragment, D is no longer applicable on this fragment. Therefore, we must consider reversing the edges before the topological sort. However, topological sorting algorithms are only applicable to directed acyclic graphs. Since conflict graphs are likely to contain cycles and vertices are weighted, only approximate algorithms exist in the literature (Al-Herz and Pothen, 2019). Nonetheless, our goal is to assign a partial order to all vertices such that applying a rule with lower order will less likely prevent the application of other rules while maximizing *filter*,*elem*, and *sem*. Therefore, we propose an algorithm (Algorithm 4) that sorts weighted vertices and edges of a directed cyclic graph based on heuristics.

Algorithm 4 Priority ordering of the vertices of a conflict graph

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

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

ToDag() transforms a graph into a directed acyclic graph by iteratively removing vertices from the strongly connected components. We implement Tarjan's algorithm (Tarjan,

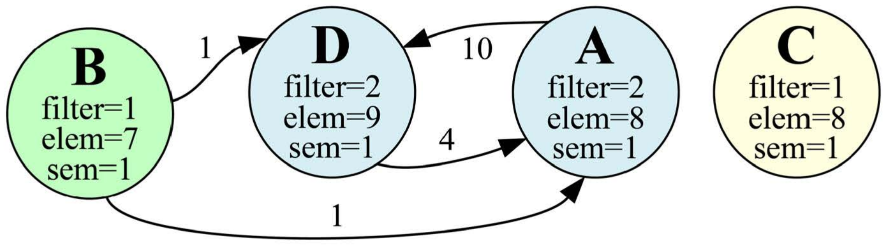


Figure 4.13. A sample conflict graph

1972) to find the strongly connected components of the graph in $O(|V|+|E|)$ time complexity. If G is the conflict graph in Fig. 4.13, then the strongly connected components are the subgraphs $\langle A,D \rangle$, $\langle B \rangle$, and $\langle C \rangle$. Thus, to make L acyclic, we should remove either vertex A or D. We define the following heuristics (in this order), to choose which vertex to remove from a strongly connected component (we denote its set of vertices by S) until L has no more cycles:

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

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

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

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

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

Hence, L contains the vertices representing rules that are less likely to prevent the application of other rules and optimize our objective. In the conflict graph of Fig. 4.13, heuristic H_1 suffices to remove A from L. All vertices of L will be given a lower priority value than vertices of R. Thus, it is important that we minimize the size of R. In our example, R consists only of vertex A. Since L is now acyclic, we apply RevTopologicalSort() to sort the vertices of L in reverse order of the edges using a $O(|V|+|E|)$ time complexity algorithm based on depth-first search. During the traversal, we use the opposite of the five heuristics whenever we have a choice between more than one vertex (*i.e.,* we minimize H_1,H_2,H_5 and maximize H_3,H_4).

On line 12, *sort* contains the sequence of vertices sorted topologically. In our example, $sort = (D,B,C,A)$ the first three from *sortedL* and the last one from *sortedR*. The algorithm constructs the *priority* function by following the order of the vertices in *sort*. However, this total order is overly conservative, *e.g.,* C has no conflict with the other rules. On lines 15–18, we ensure that if u is topologically before v and there is an edge between v and u, then $priority(v) > priority(u)$. Otherwise, they can have the same order. The *priority* function output for the conflict graph in Fig. 4.13 is presented in Table 4.1. The table also shows the initial value of the heuristics of each vertex.

Table 4.1. Priority order of the sample conflict graph in Fig. 4.13 output by the algorithm

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

When removing vertices from L to make it acyclic, we may end up with an induced graph R with the removed vertices that still contains cycles. For example, this happens if G is a complete graph, then L can only consist of one vertex that optimizes the heuristics. This is the case with the conflict graph of the Pacman game example in Fig. 4.12. Since its vertex is disconnected, `Eat` can be applied first and be part of L. All the rest of the vertices are in a clique, thus applying one would conflict with all others. However, we want to give as much chance as possible to apply as many rules as possible to optimize H_3. Nevertheless, only one of the movement rules can remain in L. According to H_1, `Move` has the highest number of conflict reasons, so it should be applied last and be part of R. All the other four vertices have the same $conf$ value. According to H_4, `Up` should have the lowest priority value among them and be part of L. Thus, all remaining rules are part of R, still forming a clique. Therefore, on line 8, we recursively order R until it is acyclic. Rules `Left`, `Right`, and `Down` are structurally very similar, except the latter which has one more element (the scoreboard). Semantically, this means that `Down` is more precise than the other two rules because it requires matching more elements. Applying another rule may risk removing this additional element, and thus not allowing `Down` to be applicable anymore. Therefore, according to H_5, `Down` should have a lower priority value than the other two rules. `Left` and `Right` rule cannot be further distinguished. Hence, any order between them will lead to the same chance of making the other inapplicable. Table 4.2 summarizes the order generated by Algorithm 4. The table also shows the initial value of the heuristics of each vertex.

Table 4.2. Priority order of the Pacman game rules output by the algorithm

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

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

5.4. Extensions

Some extensions to the heuristics we present could be considered. In particular, the goal of H_5 is to favor more precise rules as a last resort. Currently, *elem* only counts the elements to be matched in a rule. One could argue that a rule with NACs is more precise than one without, since it has fewer chances of matching. Thus it could be possible to count NAC elements in *elem*. One could also argue that a rule with abstract elements is less precise than a similar rule using one of its subclasses. For example, consider the `Move` rule in Fig. 4.11 (e). Suppose we had another rule `MoveAny` that relied on the `PositionableEntity` class instead of the `Pacman` class. Then `Move` can be considered more precise than `MoveAny`, since it has fewer chances of matching. Therefore *elem* could take into consideration abstract classes and inheritance relations.

6. Evaluation

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

6.1. Implementation

We implemented DSMCompare as an Eclipse plug-in running on the Eclipse Modeling Framework (Eclipse version 2020-09). It is available on the companion website[6].

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

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

To use DSMCompare for a given DSL, the DSL Engineer needs to perform two manual tasks. The first one is to assign an appropriate concrete syntax representation to the classes and relationships generated in the DSDiff metamodel. The engineer only needs to consider the elements prefixed with "Diff". For each Diff class, she needs to create three versions (ADD, DELETE, MODIFY) of the concrete syntax for the diff class corresponding to the original metamodel of the DSL. For example, as depicted in Figure 10, we created three additional icons representing Pacman by adding a $+ / \times / \sim$ symbol respectively. Similarly,

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

the engineer needs to create two versions (ADD, DELETE) of the concrete syntax for the diff association corresponding in the original metamodel of the DSL. The second task is to create the SDRules for the DSL. The number of SDRules to create depends on the DSL; for example, Pacman required 12 rules, Arduino (cf. Section 6.4.1) 24 rules, and Class Diagram Refactoring (cf. Section 6.4.2) 20 rules. In general, writing a SDRule is advantageous over writing the equivalent Henshin rule. The generated domain-specific editor (*e.g.,* in Figure 8) and the abstraction level that deals directly with concepts of the DSL reduce the effort compared to creating Henshin rules using generic nodes and edges, and adding explicitly graph transformation inscriptions (e.g., NAC groups as shown in Algorithm 3).

6.2. Objectives

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

RQ1 Are fine-grained differences more verbose than semantic differences?

RQ2 Does assigning priorities to semantic differencing rules yield less verbose difference models?

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

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

RQ3 Can we extract semantic differences from fine-grained diffs?

RQ4 Are semantic differences recurring?

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

6.3. Reducing the verbosity with semantic differencing

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

6.3.1. *Experimental setting*

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

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

For the MM-Refactoring case, we have specified 20 semantic difference rules adapted from the metamodel and object-oriented refactoring catalogs[7], such as Extract-Superclass, Split-References, and Rename-Attribute. The conflict graph of the rules forms two disconnected graphs. The first graph contains four rules, three of them (for method movement) forming a strongly connected component. The second graph comprises strongly connected components of 13 rules: eight for references and five for attributes. All rules filter one or two elements, except the three renaming rules, which have no filter. They are all composed of five to nine elements. As opposed to the Pac-Man case, the MM-Refactoring case represents situations where there is more variability between the semantic differencing rules.

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

Table 4.3. Results of applying the semantic differencing rules in different orders on the difference models. The numbers in the form $x \mid y$ represent x semantic difference objects and y fine-grained differences remaining in the difference model after applying all semantic differencing rules in the corresponding order.

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

Model generation. To address RQ2, we want to verify that applying DSMCompare to a difference model maximizes the number of semantic difference objects and minimizes the number of fine-grained differences. Since it is not tractable to test exhaustively all possible difference models of each DSL, we derive a representative set of difference models covering most cases. Therefore, we construct five difference models (M1 to M5) by varying the number

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

of occurrences of each rule when applied in isolation, *i.e.,* assuming there are no conflicts between rules.

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

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

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

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

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

The first four columns of Table 4.3 summarize the setup of each case. The *#fine-diffs* column shows the total number of fine-grained differences in each difference model before applying the semantic differencing rules. For instance, there are 90 fine-grained differences for M1 of the Pac-Man DSL, among which 76 are association differences and 14 are class differences. To better characterize each model, the next column (labelled *Without conflicts*) shows the total number of matches[8] of all semantic differencing rules when run in isolation,

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

assuming there are no conflicts between the rules. This gives an idea of how many times the rules are applicable; though this number is not reachable when there are conflicts between rules. For example, for M1 of the Pac-Man DSL, if all rules were to be applied on all their matches, the resulting difference model would contain 77 semantic differences, and all of the 90 fine-grained differences would be filtered. Note that the difference models for MM-Refactoring are on average 4.5 times larger than those for Pac-Man.
Priority orderings. The first independent variable of this experiment is the difference model (M1–M5) to avoid a bias in the priority order output by our approach. Furthermore, to answer RQ2, we must compare the order output by DSMCompare with other orders. One interesting order we can compare with is the reverse order. This allows the rules with most conflicts to be applied first. Other orders to compare to are obtained through random sampling from all possible permutations. However, it is intractable to test against all possible permutation of rule ordering. One property of Algorithm 4 is that rules with the same priority have no conflict between them. We denote rules sharing the same priority as a *cluster*. Thus, the order within each cluster does not have an impact on the other rules. Therefore, we can ignore the permutations within clusters. For the Pac-Man case, we obtain 6 clusters for the 12 rules and for the MM-Refactoring case, we obtain 9 clusters for the 20 rules. Still, manually testing all these possible permutations is not feasible (720 and 362 880 for the Pac-Man and MM-Refactoring cases respectively). In the random sampling, we generated orders such that no cluster has the same priority twice. Therefore, there are as many orders as there are clusters. After excluding the order output by Algorithm 4 and the reverser order, we end up with 4 additional random orders for Pac-Man and 7 for MM-Refactoring cases.

6.3.2. *Results*

Table 4.3 shows the results of this experiment. In bold, we highlighted the cases where the metrics are optimized: maximizing the number of semantic differences and minimizing the number of fine-grained differences. For example, for the M1 model of Pac-Man, applying the rules in a priority ordering output by DSMCompare results in 60 semantic differences with 30 fine-grained differences remaining. In all the tested cases, the results show that the priority order output by Algorithm 4 maximizes the number of semantic differences. Nevertheless, for Pac-Man, one of the random orderings filters more fine-grained differences than our order in three difference models. After manual inspection, we identified that this is because, in this ordering, the Pacman-Move rule has a higher priority than Pacman-Right. However, this means that a more general semantic difference takes precedence over a more specific semantic difference. This contradicts our heuristic H_5, which favors the latter over the former. This is a desirable property of our ordering since, in practice, if the Pac-Man moved to the right, then we would like that the difference model depicts the direction in which it moved.

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

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

6.4. Case studies

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

6.4.1. *Arduino Designer*

Description. Arduino Designer is an environment specially tailored to young children, to create simple programs for Arduino[9], an open-source electronics platform based on easy-to-use hardware and software. The Arduino Designer language is a DSL built to model Arduino configurations and programs graphically, based on Sirius. The DSL has two parts:

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

one for the configuration of devices and another for sketching programs. The configuration part contains primitives for placing hardware devices on the appropriate pins of the Arduino board. In Arduino, the code is placed and executed within a main loop. The sketch part models the code within the loop. It is a graphical programming language with arithmetic expressions, loops, and conditional instructions.

Just like code, these models evolve in new versions. For example, in a GitHub repository[10], we can find a history of different models that underwent bug fixes, improvements, and migrations to a new framework. Understanding complex changes that have occurred from one version to another may be hard for Arduino developers, especially if they are children. Our approach can help these developers visualize the changes in the same graphical language and environment they used for development. Furthermore, we report the changes as semantic differences. For the sketch part, we reuse known code refactoring patterns and model them as semantic differencing rules. The changes in the configuration part typically consist of adding or replacing devices in appropriate pins of the board.

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

The Arduino GitHub repository includes 13 working example projects. We filtered 6 of them, since they had an initial empty model, and just another version adding all model elements. We applied DSMCompare on all remaining 7 projects, and Table 5.1 summarizes the results. Each model has between 2 and 4 versions in the repository. The commit message associated with a version helped us to identify the purpose of the model changes (shown in the *Version n* and *Version n+1* columns). The fourth column (*Fine Diffs*) shows the total number of fine-grained differences found by DSMCompare. For example, in the `fadelight` project, when comparing the version *While* and the version *Sub instructions* (versions 1 and 2 of this project), DSMCompare reported 21 fine-grained differences. The column *Semantic*

[10]`https://github.com/mbats/arduino/`

Table 4.4. Comparison of model versions in the Arduino Designer examples repository

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

Diff Rules shows the name of the semantic differencing rules recognized among the fine-grained differences, and column *Occurrences* represents the number of occurrences of each rule. Finally, the last column shows the number of remaining fine-grained differences after some differences were removed (filtered) by applying the semantic diff rules.

Results. Table 5.1 clearly shows that DSMCompare is able to extract semantic differences from fine-grained differences, being able to report one or more semantic differences across all versions of the considered projects. Moreover, most semantic diff rules (13 out of 24, 54%) where applied several times, and 29% of them were applied across different projects.

As an illustration, for the **fadelight** project, DSMCompare reported two semantic differences of type *"Refactor a while loop"*, representing a while-loop refactoring (cf. Fig. 4.14). The first while-loop sets the device for a specific time in the *on* state, and the second loop models the *off* state of a *"FadeLight"*. In addition to one class difference, each of the two semantic diffs has also two diffs of associations. One of them represents the *"condition"* of the while-loop, and the other a link to the *"next"* instruction after the loop.

As expected, the fine-grained $Diff_{12}$ models contain fewer changes (cf. last column of Table 5.1) after applying the semantic diff rules. For example, the `tigger.tail` model adds an infrared sensor to a digital pin, and a servo motor to another digital pin in the Arduino board. The board also adds instructions to the end of the main loop. In this case, the fine-grained $Diff_{12}$ model shows the removal of six fine-grained differences and the addition of 14 fine-grained differences (a total of 20 changes). These changes can be encapsulated in

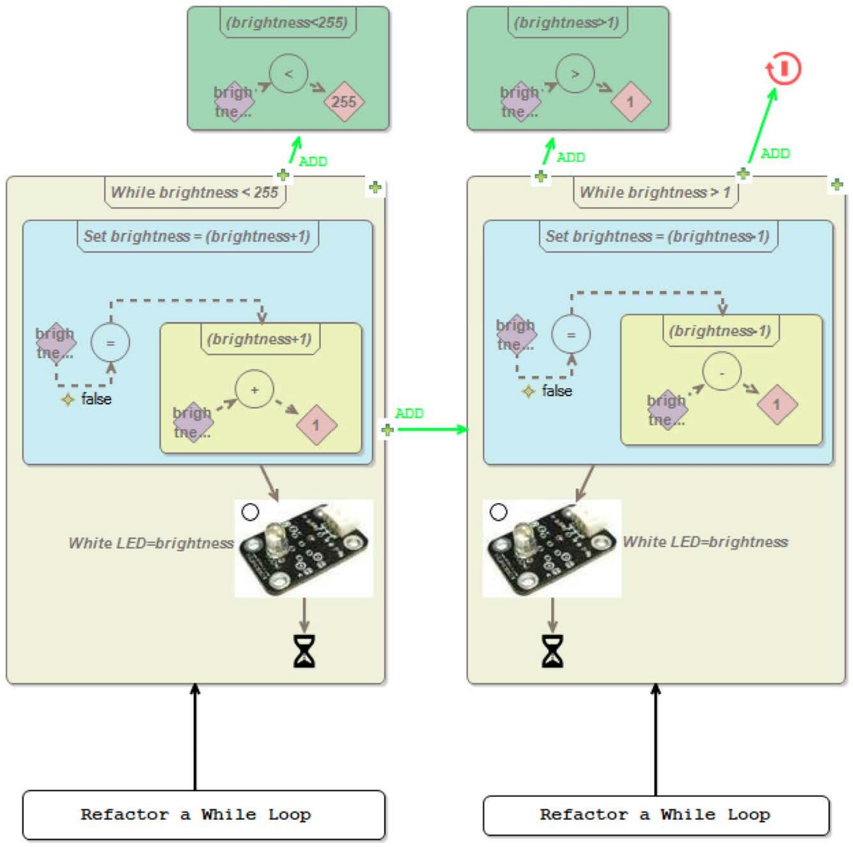


Figure 4.14. Domain-specific differences in Arduino designer for the **fadelight** project

five semantic differencing rules, i.e., *“Add an Output Module”*, *“Add Connector”*,*“Replace an If condition”*, *“Add a Status”*, and *“Move Delay”*. These rules correspond to the intention of the change *i.e., “add miaou sound to the cat”*. Meanwhile, seven fine-grained differences have been removed.

Most of the identified semantic differences in Table 5.1 are additions to an already designed Arduino model related to fix bugs, make improvements, or migrate to a new framework. Due to the nature of the Arduino DSL, any insertion of a device in the configuration part also requires changes in the sketch part. In `fadelight`, only the sketch part of the model is affected as we are inserting a while-loop to turn the LED light on and off gradually.

6.4.2. *Class Diagram Refactoring*

Description. The second case study is about refactoring class diagram models. We focused particularly on refactoring metamodels defined in Ecore from two repositories. The first repository contains three versions of the UML metamodel[11]. The second repository comes from the Graphical Modeling Framework (GMF)[12], an open source project for developing graphical modeling editors. GMF consists of two main metamodels, namely `gmfgraph` that defines the graphical notations and `gmfmapping` that maps domain models, graphical notations, and tool definitions. A description of GMF and its history can be found at (Herrmannsdoerfer et al, 2009). We extracted the metamodel versions from the version control system of GMF as previously performed in (Herrmannsdoerfer et al, 2009). The repository contains 11 versions of the `gmfgraph` metamodel and 16 versions of the `gmfmapping` metamodel. The metamodels for UML and GMF are of similar sizes with 44 classes, 61 associations, and 29 attributes on average.
Domain-specific comparison of Class Diagram Refactoring. We applied DSMCompare on the Ecore metamodel to compare different versions of the Ecore models for UML and GMF. The original metamodel *EcoreMM* consists of 20 classes, 48 associations, and 33 attributes. With DSMCompare, we generated the difference metamodel *EcoreDiffMM* with 85 classes, 165 associations, and 174 attributes. The rule metamodel *EcoreRuleMM* contains one more class and association, with 346 attributes.

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

Table 4.5 describes the consecutive model versions we considered for each project. We chose the versions that had at least two differences between each consecutive version. The semantic differencing rules we applied are the same rules as for the MM-Refactoring experiment in Section 6.3. However, the four rules related to refactoring methods (rename, pull-up, push-down, and move) are not applicable to the Ecore models. Therefore, we considered 16 semantic differencing rules.

Table 4.5. Comparison of model versions for class diagram refactoring from different repositories

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

Results. Table 4.5 reports the results of applying DSMCompare in a similar fashion as for the Arduino case study. For example, the fine-grained Diff_{12} model between the Ecore models of `gmfgraph` versions 1.29 and 1.30 reports 59 fine-grained differences. We found nine applicable semantic differencing rules. Among them, the *"extractSuperClass"* rule, which removes attributes from a class and creates a new parent class containing these attributes, occurs in three consecutive model versions, on a total of five matches.

Like for the Arduino case, we see that the fine-grained Diff_{12} model contains fewer changes (cf. last column in Table 4.5) after applying the semantic differencing rules. For example, in the UML project, Diff_{12} shows that one association is added, while another one is removed, and the type of the association is modified. These fine-grained changes can be encapsulated in the SDRule *"mergeReference"* which corresponds to the intention of the change. As a result of this rule application, fine-grained differences are filtered.

Table 4.5 shows that DSMCompare is able to extract semantic differences from fine-grained differences. In some cases, it reports more than one match of the same semantic difference, *e.g.*, *"moveAttribute"* (10 times in `UML`) or *"specializeSuperType"* (6 times in `GMFgraph`). In the latter case the rule filters a fine-grained difference at every match, thus presenting less irrelevant information to the user. However, not all rules have filters. For example, *"replaceEnum"* does not filter elements. Nevertheless, by adding semantic difference objects, DSMCompare lifts the user's understanding of changes closer to her intentions: attributes are replaced by enumerations. DSMCompare is also able to find the expected semantic differences (according to the commit messages). For example, it detected the *"extractSuperClass"* rule in both the `GMFgraph` and the `GMFmappings` projects.

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

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

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

The box-plot in Fig. 4.15 reports standard descriptive statistics that can be read as follows: the lower bound of the rectangle is the first quartile, the upper bound is the third

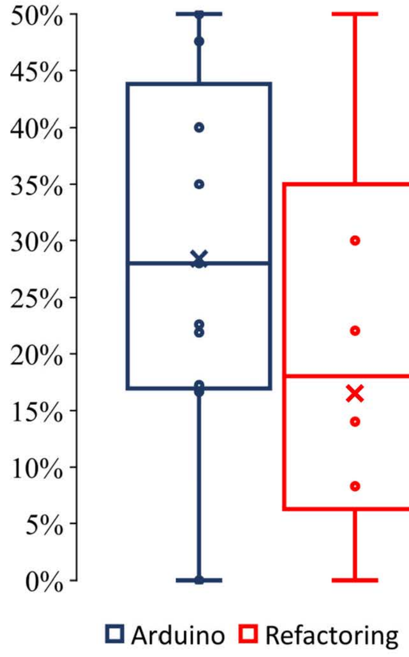


Figure 4.15. Verbosity reduction for each case study

quartile, the middle bar within the rectangle is the median, the cross is the average, and the top and bottom vertical lines denote the amplitude of the data. Fig. 4.15 reports averages (28% and 16%), medians (28% and 18%), and standard deviations (17%) for both case studies respectively. ArduinoDesigner models contain fewer elements and the differences reported fewer fine-grained differences, which may explain higher verbosity reductions overall. We note that $VR \in [0{,}0.5]$ in these projects; thus, using SDRules reduces the fine-grained differences reported by up to a factor of two.

Fig. 4.16 reports the ratio of SDRules per occurrence. For example, 17% of the Arduino and Refactoring SDRules recur four times. We counted a rule as recurring if it matches multiple times on the same model or if it is present in multiple versions. We notice that SDRules occur multiple times and across different projects in each case study. On average, each rule occurs around three times. Also, the majority of the SDRules occur at least twice for each case study (52% for Arduino and 61% for Refactoring). This justifies that the chosen rules are appropriate for these DSLs.

6.5. Discussion

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

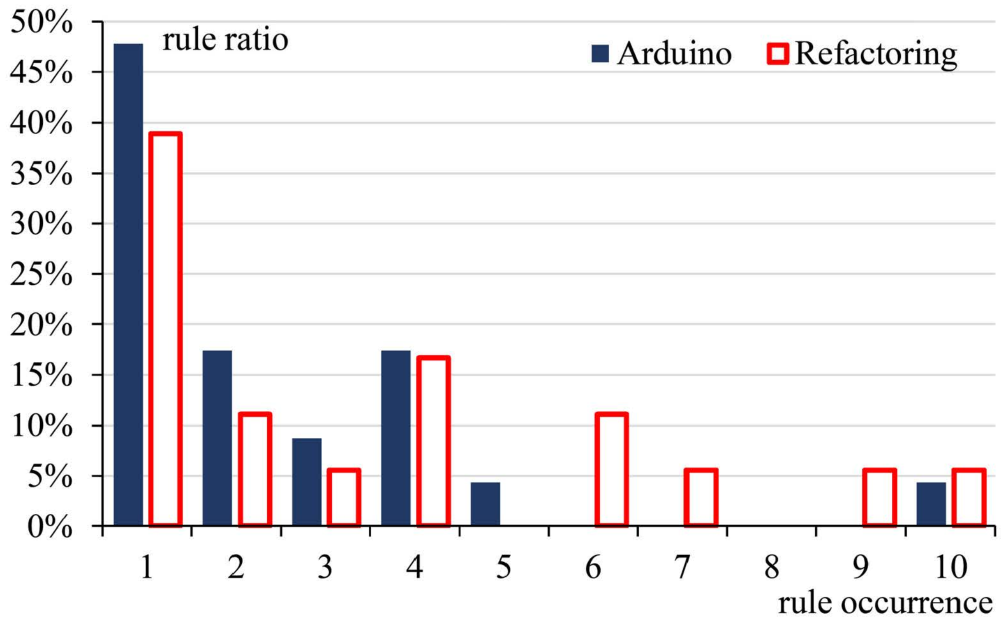


Figure 4.16. Ratio of the number of rules per number of occurrences for the two case studies

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

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

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

Ability to extract semantic differences (RQ3). The premise of this work relies on the ability to report semantic differences in the difference models. The case studies, based on models developed by third-parties, validate that it is possible to find such semantic differences. In DSMCompare, semantic differences are specified by semantic differencing rules. If every rule only occurred once in the case studies, then they would be too specific for each difference model which means they would have to be specified by the end-users almost every time they use DSMCompare. However, our results have shown that the rules we have extracted from the case studies occur multiple times (cf. Fig. 4.16). This strengthens the view that it is possible to extract semantic differences in practice and that the rules can be specified only once per DSL. Nevertheless, with more Diff_{12} models, the set of SDRules may grow. Domain experts can modify or add more rules incrementally thanks to the generated SDRule editor that uses the same environment and notations as the modeling editor used for the DSL.

Applicability of semantic differencing rules (RQ4). We found that SDRules occur in different Diff_{12} models (cf. Fig. 4.16). This means that they are not specific to a given Diff_{12} model, but generally applicable to any Diff_{12} model of the DSL. The semantic difference rules must come from a piece of knowledge within the DSL. Therefore, they cannot be invented and need to be semantically meaningful. This knowledge can originate from the operational semantics if it is an executable model, or refactoring patterns, or some DSL-related knowledge-base. From our experience with DSMCompare, SDRules come from the operational semantics of the DSL (as in Pacman) or known refactoring patterns in the DSL: *e.g.,* code-level (like the sketch in Arduino), class-level (like in Ecore models), or model-level (like in feature models (Tanhaei et al, 2016)). Additionally, in practice, when the DSL designer observes that specific differences are often recurring, this is a good indicator that this may be elevated to a semantic difference rule, to simplify the comprehension of the diffs by the DSL users. Hence, overall, DSMCompare supports both top-down (i.e., based on known refactorings of the DSL, or its semantics) and bottom-up (i.e., mined from actual changes) approaches to specify SDRules. Finally, our approach is agnostic of the meta-level of the input artifact. As demonstrated in the evaluations, the input can be the metamodel of a DSL (like Pacman and Arduino) or it can be a meta-metamodel (like Ecore).

6.6. Threats to validity

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

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

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

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

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

7. Related work

This section reviews related works on model differencing. The survey in (Stephan and Cordy, 2013) presents several model comparison approaches and applications. Model differencing involves *calculation* of the matching model elements, *representation* of their differences, and *visualization* of the differences. Hence, we structure this section paying attention to these three aspects, and also review control version systems for modelling artefacts.

Model matching calculation. Kolovos et al. (Kolovos et al, 2009a) survey current approaches for model matching. These can be: *static identity-based*, which assume a unique identifier for objects; *signature-based*, which compare objects based on a dynamic signature calculated from the objects' properties; *similarity-based*, which match objects based on the weighted similarity of their properties, but obviates the model semantics; and *language-specific*, developed ad-hoc for a modeling language and its semantics. For example, using *signifiers* (Langer et al, 2012) (*i.e.,* combinations of features of a metamodel class) as a comparison criterion falls in the signature-based category, EMFCompare is similarity-based but permits defining custom matching algorithms, and UMLDiff (Xing and Stroulia, 2005) is language-specific. In general, each solution is a better fit for certain kinds of problems: a language-specific matching algorithm may be faster and more accurate than a generic algorithm, but its implementation requires more effort.

Maoz et al. (Maoz et al, 2011a) argue that existing model differencing approaches are purely syntactic and challenge the community to develop semantic diff operators. These calculate a set of diff witnesses that give proof of the real change between two models and the effect on their semantics. Two models may be syntactically different but have no diff witnesses, meaning that they are semantically equivalent. For example, a diff witness of two class diagrams would be an object diagram that is an instance of one of the class diagrams but not of the other, while for activity diagrams, it would be an execution trace admitted by only one of the diagrams. Diff witnesses also allow deciding whether the semantics of two versions of a model are equivalent, incomparable, or one refines the other. This approach was later realized in the Diffuse framework (Maoz and Ringert, 2018). Extending our approach to deal with model diffs concerned with the instantiability or executability of models as a comparison criterion is left for future work.

Some researchers have dealt with N-way matching (Holthusen et al, 2014; Reuling et al, 2019), especially in the context of extracting a product line out of a set of structurally similar model variants. In this case, N-way matching is needed to identify the common parts of the involved artefacts. We plan to extend DSMCompare to capture changes between more than two models, and so in this context, it could be used to better understand the (semantic) differences between several model variants.

Representation of model differences. Cicchetti et al. (Cicchetti et al, 2007) propose an approach to represent model differences that is metamodel independent and agnostic of the difference calculation method. Specifically, given two models conforming to the same metamodel, their difference is expressed as another model that conforms to a new metamodel.

This new metamodel is derived from the original one by a transformation and allows representing model changes (additions, deletions, and changes). Such difference models induce transformations to translate from one model version to the other and can be composed. While this approach to represent model differences is similar to our proposal, it only works at the abstract syntax level, whereas we also deal with the concrete syntax and support domain-specific patterns to visualize the model differences.

Our approach extends the metamodel of the DSL to represent semantic differencing rules for domain-specific model differences. A related technique is the ramification of metamodels for domain-specific model transformations (Kühne et al, 2009). In this approach, graph transformation rule patterns are expressed in a domain-specific way. The metamodel of the patterns is generated by transforming the metamodel of the input/output DSLs: relaxing cardinalities, adding transformation-specific attributes and other concepts, and modifying attribute types.

Since low-level differences returned by generic comparison tools may be difficult to understand, Kehrer et al. (Kehrer et al, 2011) perform a semantic lifting of such differences to the level of editing operations. For this purpose, low-level differences are represented as models, so that the identification of editing operations consists of finding groups of related low-level changes. This search is performed by rules that are automatically derived from the rule-based specification of the editing operations. Hence, the notion of semantic lifting is similar to our rules for expressing domain-specific semantic differences. However, semantic lifting only deals with the abstract syntax of models, whereas we consider the concrete syntax as well. Similar to semantic lifting approaches such as (García et al, 2013; Vermolen et al, 2012) we identify complex change patterns from low-level changes involved in a metamodel evolution. Although these patterns resemble the rules in our approach, they are generic and predefined. In contrast, our approach allows the DSL engineer to define the semantic differencing rules.

Visualization of model differences. Gleicher (Gleicher, 2018) provides general guidelines for visualizing comparisons. For many different domains, comparing artifacts is a common task and visualizing the comparison often helps. Generally, the visual comparison is displayed using juxtaposition (*e.g.,* as EMFCompare does in Fig. 4.3), superposition, or explicit encoding (like we do in Fig. 4.10).

Brosch et al. (Brosch et al, 2012b) visualize the changes and conflicts in concurrently evolved versions of the same UML model using UML profiles (stereotypes and tagged values). This permits modelers to resolve the conflicts within the UML editor of their choice while using the concrete syntax of the manipulated language. However, this approach is only suitable for UML models whereas we pursue a general approach for arbitrary domain-specific languages.

More similar to our work, the authors in (Schipper et al, 2009) focus on the visualization of diagram differences in the diagrams themselves. The rationale is to help users to understand the modifications immediately. Their proposed visualization includes pop-ups reporting the changes performed in the neighborhood, zooming to changes, collapsing irrelevant parts, and using different colors to represent additions (green), deletions (red), and changes (blue), either in a single diagram or confronting two diagram versions. They have developed a tool that uses EMFCompare for model comparison, as we do. However, their tool only permits visualizing atomic changes, represented by different colors. Instead, we support both fine-grained and coarse-grained domain-specific patterns of change. Furthermore, the visualization associated with each pattern is highly configurable. Other works, such as (Mehra et al, 2005; Ohst et al, 2003), only permit showing changes using different colors or shape styles.

A few works deal with the scalable visualization of differences in the case of large models. To solve this problem, van den Brand et al. (van den Brand et al, 2010) combine a generic visualization framework for metamodel-based languages to show the fine-grained differences, with polymetric views that provide support for zooming and filtering. Wenzel (Wenzel, 2008) also relies on polymetric views to support scalable visualization of differences based on model metrics. Both works are complementary to ours: whereas we provide domain-specificity to the visualization, these other works add a general visualization layer on top.

Version control systems for models. Even though models are frequently persisted as text files, the use of traditional text-based version control systems is suboptimal, as we have argued in the introduction. This way, several model repositories with support for version control have been proposed along the years (Altmanninger et al, 2009). The ModelCVS (Kappel et al, 2006) and the AMOR projects (Altmanninger et al, 2008a) proposed dedicated version control systems for models with sophisticated functionalities, like a recommender of possible resolutions for model conflicts (Brosch et al, 2010d). In this setting, DSMCompare could be useful to help understand better the differences between the models, before choosing a resolution strategy.

The model repository of Espinazo-Pagán and García-Molina (Espinazo-Pagán and García-Molina, 2010) uses a MySQL database for storage, and a special encoding of model versions to improve efficiency. For a better performance, the authors later proposed the use of NoSQL databases for persistence (Espinazo-Pagán et al, 2011). EMFStore (Koegel and Helming, 2010b) and CDO (CDO Model repository, accessed August 2023) are well-known model repositories for EMF, which support collaborative editing and versioning of models. DSMCompare could be used atop these repositories to enable the visualization of (semantic) diffs using the graphical concrete syntax of the DSL.

Commercial modeling tools feature different levels of versioning and model differencing capabilities. LabView has a built-in revision control system that allows to programmatically compare different models (LabView, 2023). MetaEdit+ (Kelly et al, 1996) features a version control mechanism called Smart Model Versioning (MetaEdit, last accessed 2023), which allows comparing models – graphically, textually or by means of a tree – and storing them on any major version control system such as Git. MPS (MPS, last accessed 2023b) integrates with Git and Subversion and provides some capabilities for viewing model differences, in a textual way (MPS, last accessed 2023a). Simulink supports comparing models and highlighting the differences in the original models. Simulink uses a scoring algorithm to determine if two model elements are a match (Simulink, 2023). Similarly, SystemWeaver (SystemWeaver, 2023) provides versioning capabilities at the model element level. This way, users can compare an element, view its history, and replace one version of an element with another. While these tools offer different ways to diff models, these are typically fixed and not customizable. Instead, our approach could be valuable here to provide domain-specific, customizable visualizations of the model differences, in a graphical way.

Our approach is based on Eclipse Modeling Framework (EMF). This is a relevant technology, since Eclipse is widely used in MDE research and many companies use Eclipse and EMF tools (Akdur et al, 2018). Large companies such as IBM are spearheading MDE through EMF (Mohagheghi et al, 2013).

Model differencing and collaborative modeling can lead to clones and duplicates. Some approaches have addressed this problem. Störrle has developed a number of heuristics and algorithms to detect clones in models (Störrle, 2010, 2017). Babur *et al.* (Önder Babur et al, 2019) leveraged natural language processing, feature extraction and clustering techniques to detect clones in models. We have not focused on detecting model clones in our approach, which is left as future work.

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

8. Conclusion

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

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

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

On the tooling side, we will improve the visualization of differences organizing them in layers (*e.g.,* to hide fine-grained differences and visualize only semantic differences). We also plan to incorporate our approach within model repositories, like MDEForge (Basciani et al, 2014), or version control systems for code, like GitHub.

Chapter 5

From two-way to three-way: domain-specific model differencing and conflict detection [1]

Manouchehr Zadahmad Jafarlou,
Eugene Syriani and Omar Alam
DIRO, Université de Montréal

This paper represents the work I did for the second contribution of my thesis.
The co-authors are my supervisors who contributed to the writing.

[1]The content of this chapter has been submitted to the The Journal of Object Technology (JOT). It is under second revision.

Résumé. Dans le travail collaboratif, les développeurs font évoluer leurs modèles en parallèle, ce qui entraîne des différences et des conflits substantiels. Pour mieux consolider ces changements, les développeurs doivent comprendre les différences en termes de syntaxe et de sémantique des modèles. Malgré une myriade d'efforts, les systèmes de contrôle de version et les outils de comparaison de modèles existants se concentrent sur les modèles génériques, sont difficilement adaptables à un langage spécifique à un domaine (DSL) et présentent principalement des changements syntaxiques au développeur. De plus, ils signalent les différences et les conflits de modèles spécifiques à un domaine sur la base de leur syntaxe abstraite plutôt que de la syntaxe concrète du DSL. Pour résoudre ces problèmes, nous avons précédemment introduit DSMCompare pour détecter les différences fines et sémantiques entre les paires de versions de modèle et présenter les changements dans la syntaxe concrète du DSL. Dans cet article, nous avons encore amélioré notre pratique en considérant une comparaison de modèles à trois voies, typique dans le contexte des systèmes de contrôle de version. DSMCompare peut désormais signaler les différences provenant des deux versions ainsi que les conflits. Pour détecter les différences et les conflits sémantiques, notre approche s'appuie sur la spécification par l'ingénieur DSL de modèles de différenciation sémantique dans un éditeur adapté au DSL. Pour évaluer DSMCompare, nous avons procédé à une rétro-ingénierie de l'historique des validations de plusieurs projets open source dans lesquels des modifications de refactorisation de code basées sur Java se produisent. Nous montrons que DSMCompare trouve efficacement ces différences et conflits sémantiques avec une grande précision.

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 and 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 (Kappel et al, 2006; Altmanninger et al, 2008b; Brosch et al, 2010a; Koegel and Helming, 2010b), 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 and 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 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

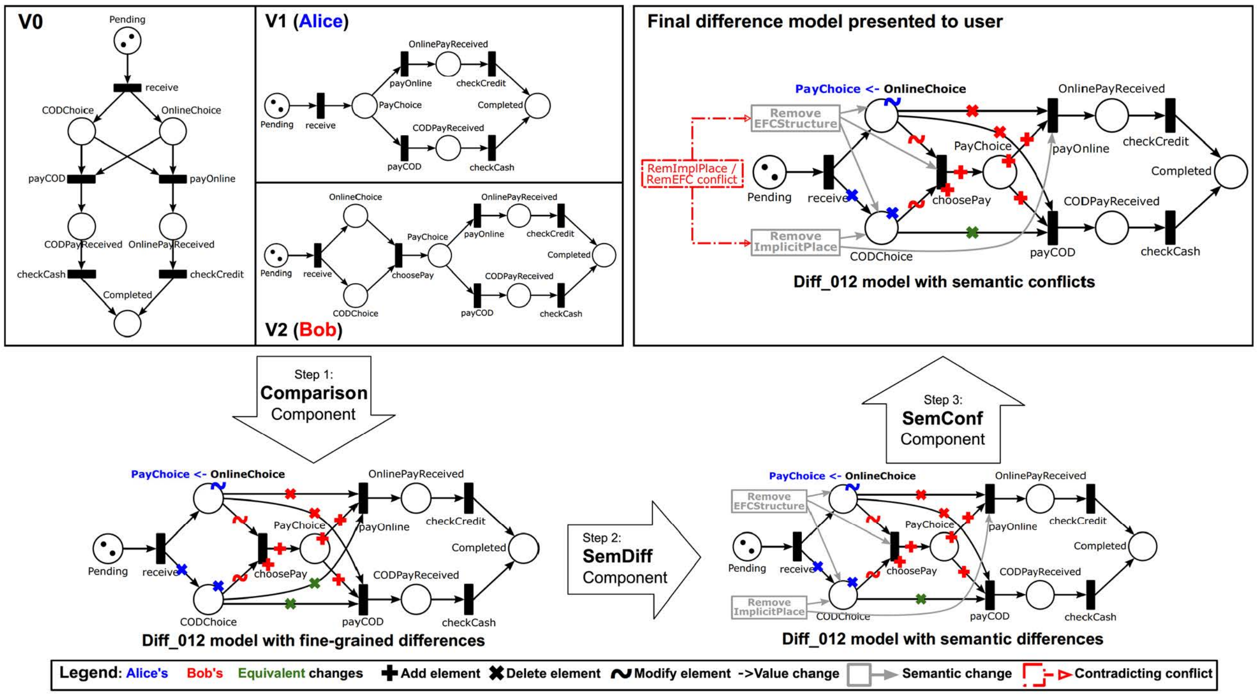

V0
V1 (Alice)
V2 (Bob)
Final difference model presented to user
Pending
receive
CODChoice
OnlineChoice
payCOD
payOnline
CODPayReceived
OnlinePayReceived
checkCash
checkCredit
Completed
PayChoice
choosePay
PayChoice <- OnlineChoice
Remove EFCStructure
Remove ImplicitPlace
RemImplPlace / RemEFC conflict
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 5.1. Domain-specific three-way comparison of WN with DSMCompare

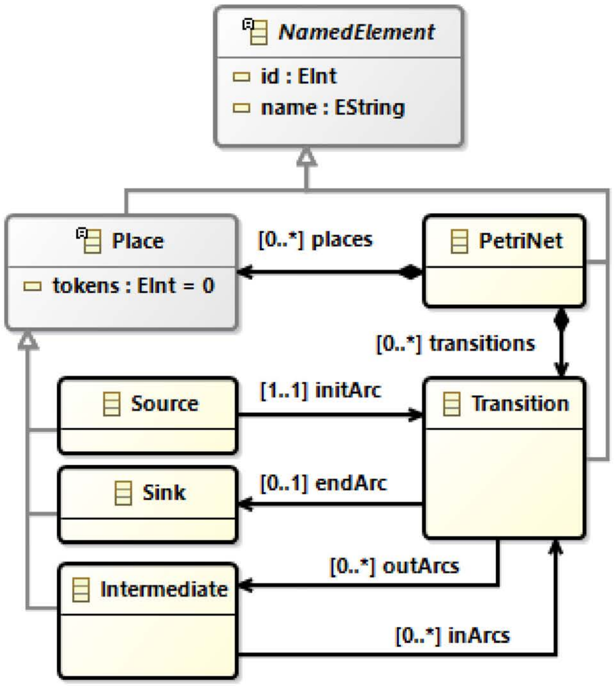


Figure 5.2. Metamodel of the Workflow WN DSL

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 Fig. 5.1 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 (Sirius, 2023a). Fig. 5.2 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 and Pierantonio, 2008) and EMF DiffMerge (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` representation 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 $+$ / $\times$ / $\sim$

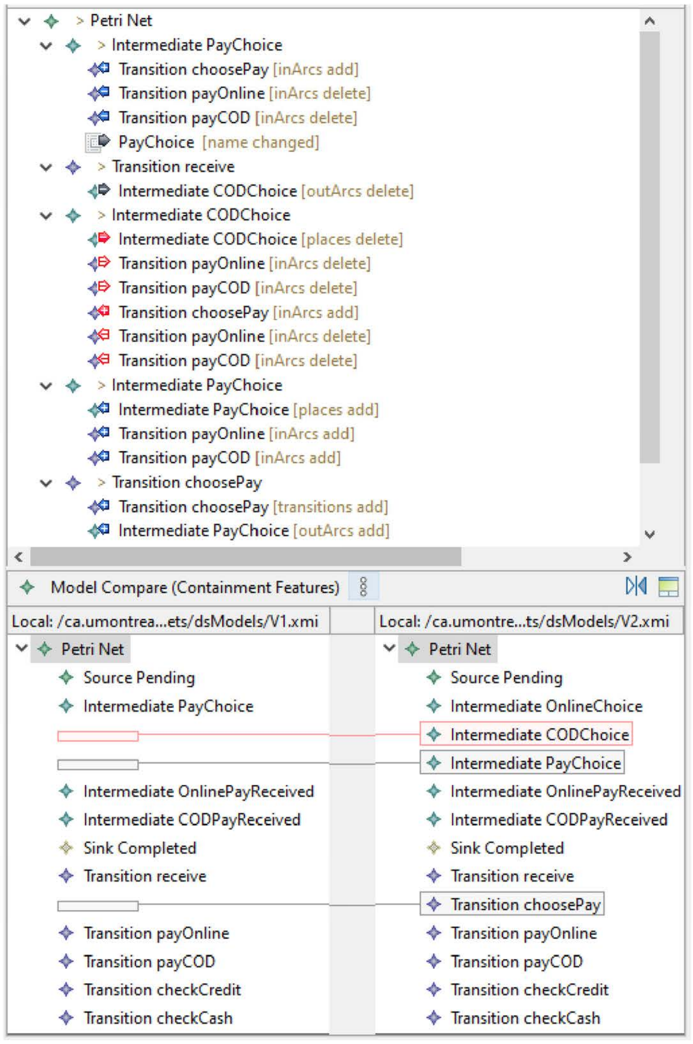
symbol to represent additions, deletions, and modifications, respectively. Additionally, WN-Compare offers a default concrete syntax to depict semantic differences, which pertains to editing semantics (*i.e.,* we do not map to the semantic domain as in Maoz et al (2011b)). Charlie can modify these graphics 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 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 Fig. 5.1 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. Fig. 5.4 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 Fig. 5.1. 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.



```
v > Petri Net
  v > Intermediate PayChoice
        Transition choosePay [inArcs add]
        Transition payOnline [inArcs delete]
        Transition payCOD [inArcs delete]
        PayChoice [name changed]
  v > Transition receive
        Intermediate CODChoice [outArcs delete]
  v > 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]
  v > Intermediate PayChoice
        Intermediate PayChoice [places add]
        Transition payOnline [inArcs add]
        Transition payCOD [inArcs add]
  v > 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 5.3. Differences and conflicts reported by EMFCompare for the running example

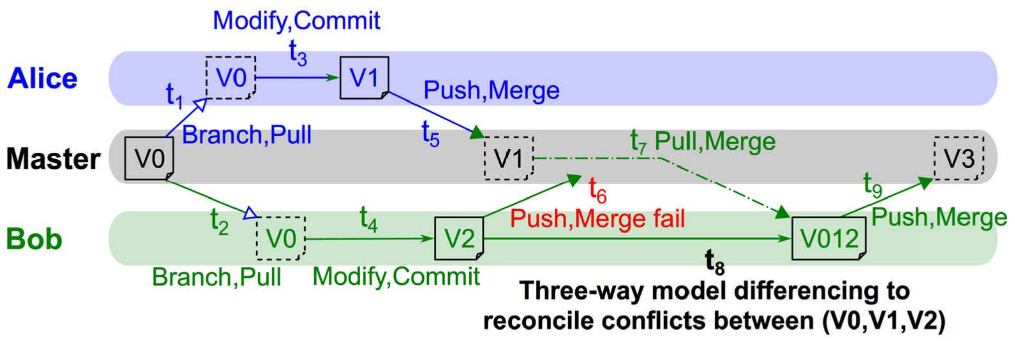


Figure 5.4. Collaboration scenario where three-way differencing is needed

Alice and Bob branch from this version and build versions V1 and V2, respectively shown at the top of Fig. 5.1. 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 Fig. 5.3. 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 Fig. 5.1, 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 et al, 2012e). 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 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 and Cordy, 2013), such as EMFCompare (EMF Compare, accessed August 2023), DiffMerge (DiffMerge, 2023), DSMCompare (Zadahmad et al, 2022), Maudeling (Rivera and Vallecillo, 2008), DSMDiff (Lin et al, 2007), and Epsilon-based Three-way Merging Process (E3MP) which uses the Epsilon Comparison Language (Sharbaf and Zamani, 2020; Sharbaf et al, 2022a).

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 and 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.

Table 5.1. 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	○	○	●	●	●	⊛

3.1. 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.

3.2. 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.

3.3. 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.

3.4. 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.

3.5. 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.

3.6. 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.

3.7. 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.

3.8. 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.

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.

3.9. 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.

3.10. 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.

3.11. 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.

3.12. 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 5.1 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, last accessed 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 and Zamani, 2020; Sharbaf et al, 2022a). 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 subcategory 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 supports semantic differences.

As DSMCompare is the only tool that fully supports two-way domain-specific differencing (comparison), we chose this tool to transform a two-way differencing tool into a three-way differencing tool in our study. 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 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 *C*1 and *C*2 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

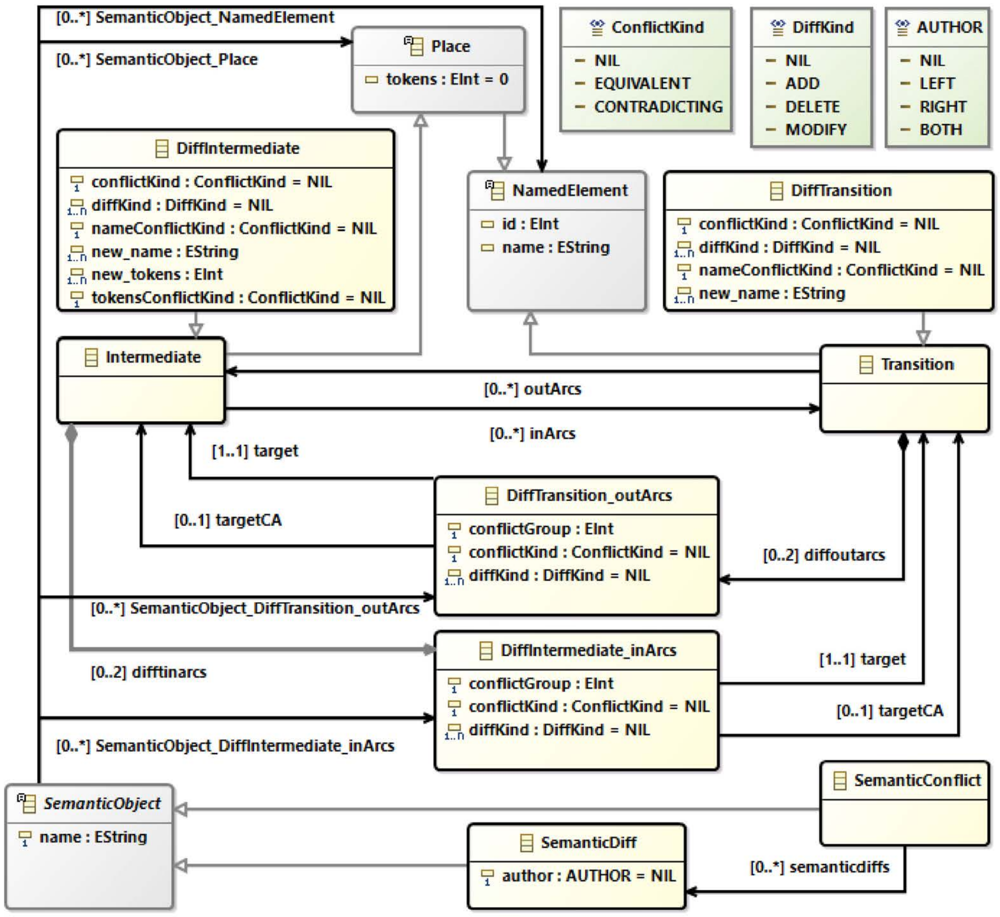


Figure 5.5. An excerpt of three-way Domain-Specific Difference metamodel (DSDiffMM_3Way)

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 Fig. 5.2. Fig. 5.5 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 Fig. 5.5 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 Fig. 5.5. 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 Fig. 5.5. *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 Fig. 5.5 must also keep track of any semantic difference since conflicts can occur between fine-grained and semantic differences, like after step 3 of Fig. 5.1. 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 Fig. 5.1, we organize DSMCompare into three components. Here, we discuss the `Comparison` component (Step 1 in Fig. 5.1). 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, Fig. 5.7a 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, Fig. 5.7a 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 and 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.

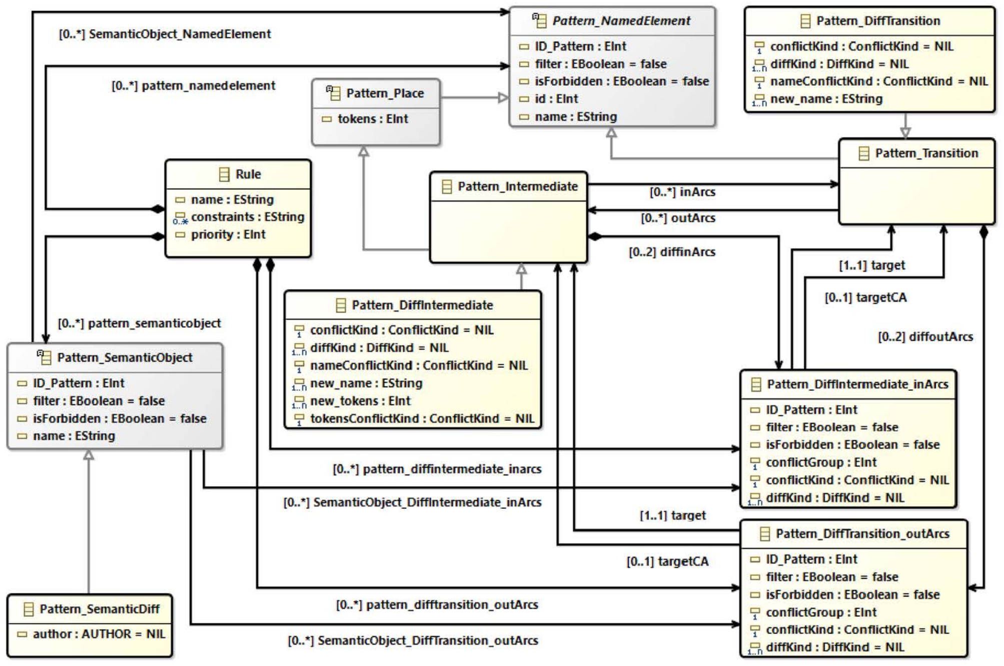
As multiple `SDRules` may be applicable simultaneously, DSMCompare uses a heuristic-based algorithm to schedule their application order. The priority order aims to reduce the verbosity of the presented differences and maximize the presence of semantic differences over fine-grained differences. Finally, 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

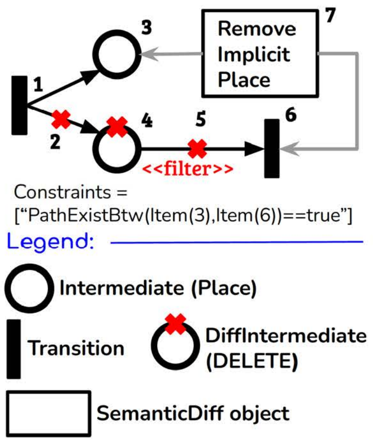
The `SemDiff` component automatically generates the rule metamodel `SDRuleMM` from the `DSDiffMM` metamodel. Fig. 5.6 shows a fragment of the result for the WN example, which extends the process outlined in Section 5.2. One improvement in the generation process is that the `Rule` root class of the `SDRuleMM` 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. Fig. 5.7a 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 Fig. 5.7b. The

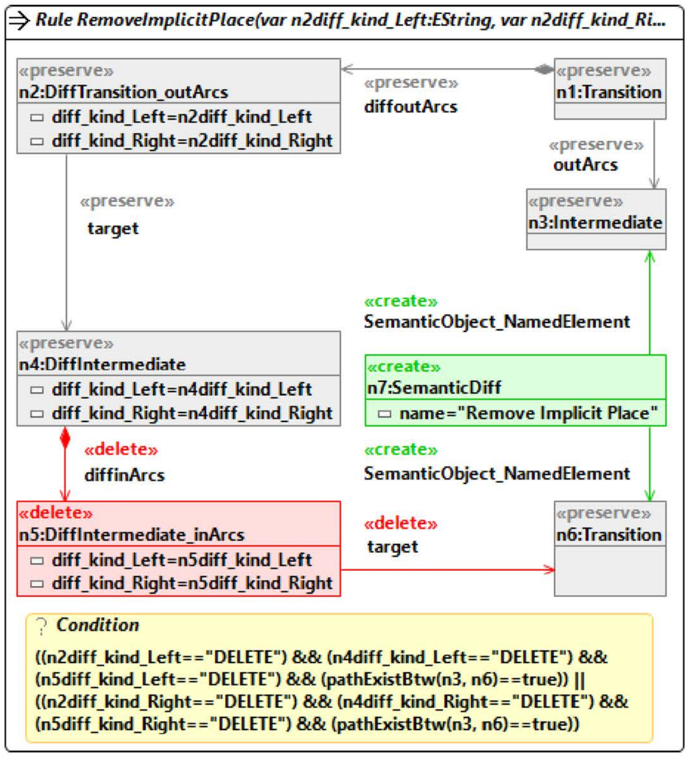


Pattern_NamedElement
ID_Pattern : EInt
filter : EBoolean = false
isForbidden : EBoolean = false
id : EInt
name : EString
Pattern_DiffTransition
conflictKind : ConflictKind = NIL
diffKind : DiffKind = NIL
nameConflictKind : ConflictKind = NIL
new_name : EString
Pattern_Place
tokens : EInt
Pattern_Transition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

Figure 5.6. 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»
n3:Intermediate
«preserve»
target
«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 5.7. A semantic difference rule transformed into a Henshin graph transformation rule

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 Fig. 5.7.

(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.

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, 2023b), 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 `BorderedNode-Mapping` 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 Fig. 5.1.

Assume that the diff class DiffA_S corresponds to the association with an incoming composition diffS 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 5.1. 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 Fig. 5.1, 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 Fig. 5.5) 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

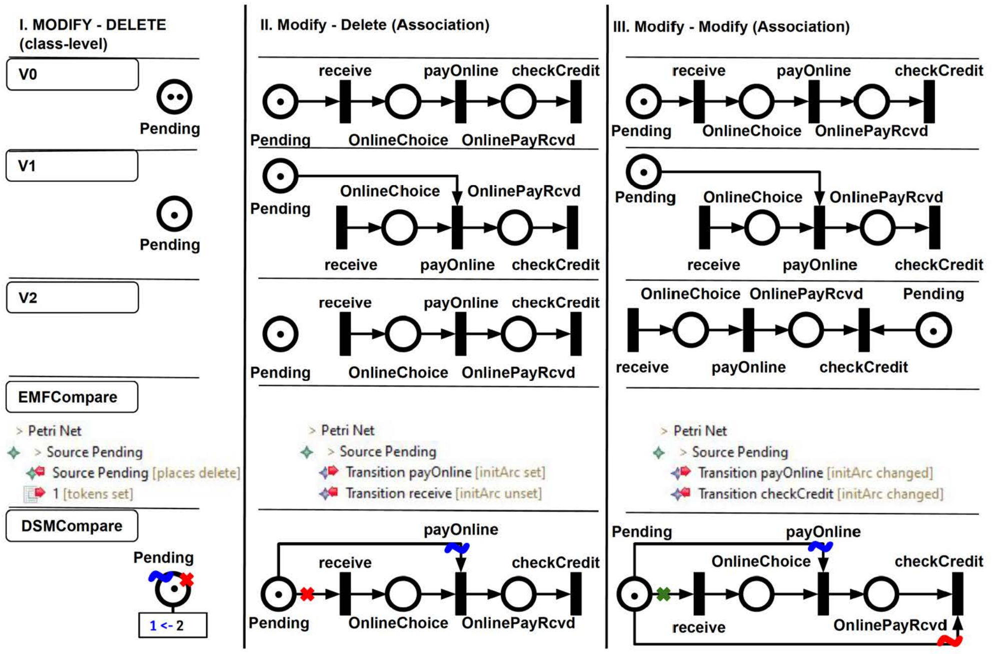
A contradicting conflict occurs when V1 and V2 have made different changes to the same element with respect to V0. Fine-grained contradicting conflicts arise when the `diffKind` array of a specific element has either *MODIFY/MODIFY* or *MODIFY/DELETE* values. A fine-grained change may also conflict with 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 Fig. 5.1, 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 Fig. 5.8 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*.



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 5.8. Examples of fine-grained contradicting conflict

The second column in Fig. 5.8 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 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 Fig. 5.5) to both diff objects representing the links.

The last column in Fig. 5.8 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

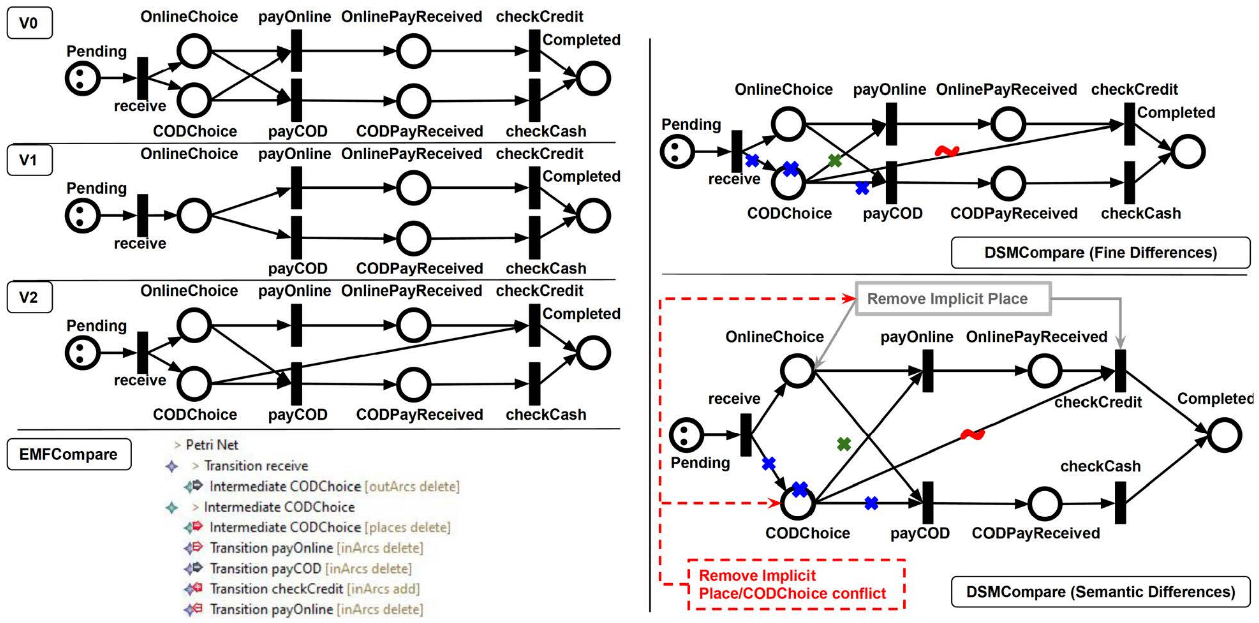

V0
V1
V2
Pending
receive
OnlineChoice
CODChoice
payOnline
payCOD
OnlinePayReceived
CODPayReceived
checkCredit
checkCash
Completed
EMFCompare
> Petri Net
> Transition receive
Intermediate CODChoice [outArcs delete]
> Intermediate CODChoice
Intermediate CODChoice [places delete]
Transition payOnline [inArcs delete]
Transition payCOD [inArcs delete]
Transition checkCredit [inArcs add]
Transition payOnline [inArcs delete]
DSMCompare (Fine Differences)
Remove Implicit Place
Remove Implicit Place/CODChoice conflict
DSMCompare (Semantic Differences)

Figure 5.9. Example of contradicting semantic-fine conflict

between fine-grained differences, conflicts involving semantic differences can overlap across multiple elements and must be carefully identified. For example, in Fig. 5.9, 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 Fig. 5.1, 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 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 5 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 Fig. 5.9, 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.

Algorithm 5 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 ← $\emptyset$
 6:     for all pc in potConflicts do
 7:       fine ← FindInstanceOf(sem,pc,diff012)
 8:       if fine.conflictKind = CONTRADICTING then
 9:         actualConflictList ← actualConflictList $\cup$ {fine}
10:       end if
11:     end for
12:     if |actualConflictList| > 0 then
13:       semConflict ← CreateSemConflict(sem)
14:       for all fineConflict in actualConflictList do
15:         CreateAssociation(semConflict, fineConflict)
16:       end for
17:     end if
18:   end for
19: end procedure
```

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 5 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 5 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 5, it then creates a `SemanticConflict` object and links it to the semantic and the fine-grained difference objects. The first row in Fig. 5.9 illustrates this situation.

8.4. Computing actual conflicts for semantic-semantic differences

Algorithm 6 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 \land fine1.conflictKind = $CONTRADICTING then
 9:         semConflict ← CreateSemConflict(sem1,sem2)
10:         CreateAssociation(semConflict,sem1)
11:         CreateAssociation(semConflict,sem2)
12:       end if
13:     end for
14:   end for
15: end procedure
```

Algorithm 6 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 5, 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 Fig. 5.1. 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 Fig. 5.1.

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[2]. To find the generic model-based matches and differences, we instantiate the CDOCompare engine[3], one of the default EMFCompare engines 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[4] to transform domain-specific rules into Henshin textual format. Then, using Henshin's 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.

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

9.2. Objectives

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

Fig. 5.10 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

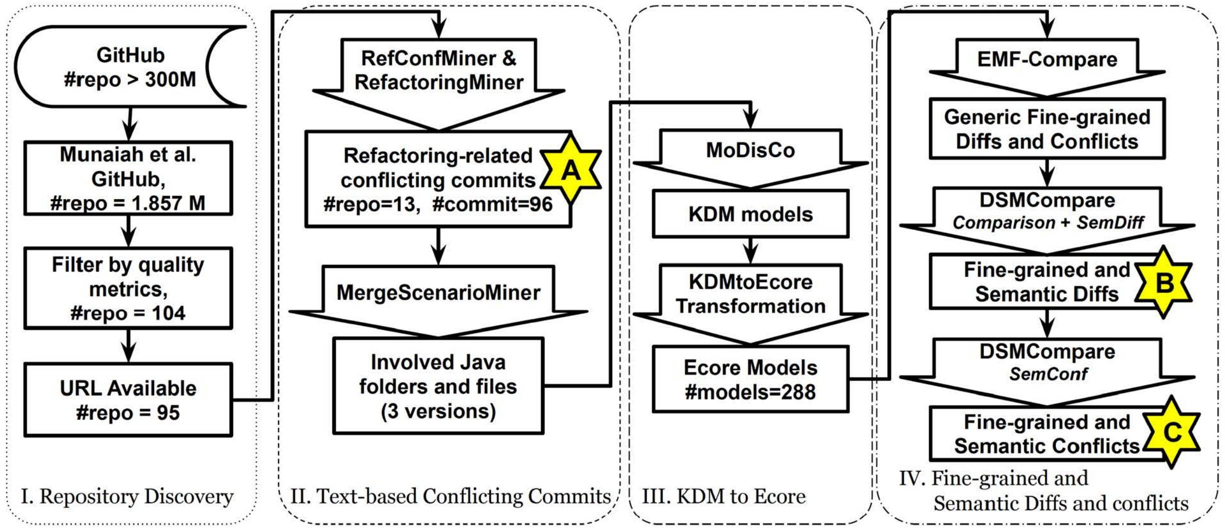

GitHub #repo > 300M
Munaiah et al. GitHub, #repo = 1.857 M
Filter by quality metrics, #repo = 104
URL Available #repo = 95
I. Repository Discovery
RefConfMiner & RefactoringMiner
Refactoring-related conflicting commits #repo=13, #commit=96
A
MergeScenarioMiner
Involved Java folders and files (3 versions)
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 5.10. Evaluation setup to execute DSMCompare on Java code from GitHub repositories

projects, reports having over 42 million public repositories[5] 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.

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

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

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 Fig. 5.10 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.

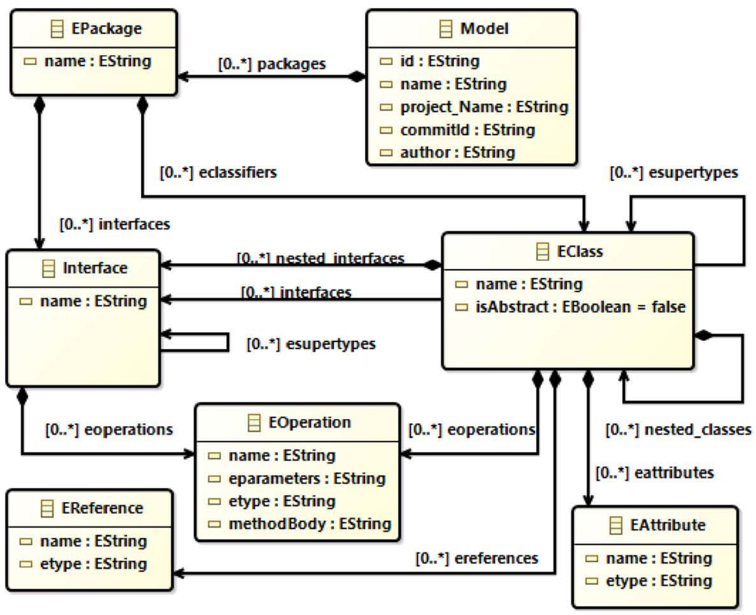


Figure 5.11. The *MiniJava* metamodel

We built a simplified MiniJava metamodel in Ecore, shown in Fig. 5.11, 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 Fig. 5.10. 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. Fig. 5.12 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

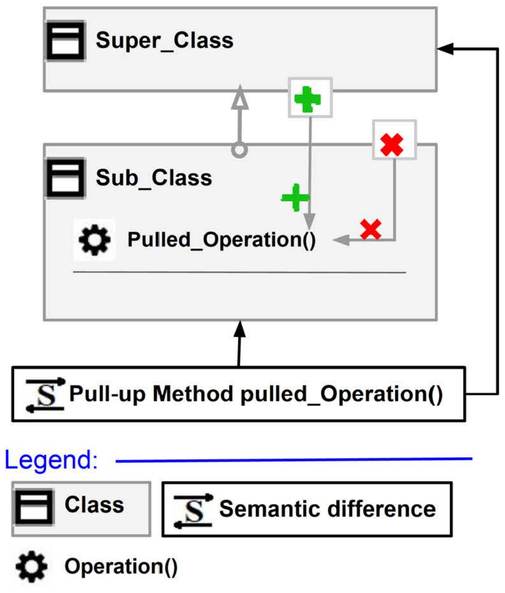


Figure 5.12. The *Pull-up Method* semantic differencing rule

by label B in Fig. 5.10. Finally, we pass the *Diff*$_{012}$ model produced by the `SemDiff` component through the `SemConf` component and populate *Diff*$_{012}$ model with semantic conflicts (label C in Fig. 5.10).

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 (9.1), 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*.

$$\begin{aligned} Precision_{diff} &= \sum_{i=1}^{96} \frac{|AD_i| \cap |B_i|}{|B_i|} \\ Recall_{diff} &= \sum_{i=1}^{96} \frac{|AD_i| \cap |B_i|}{|AD_i|} \end{aligned} \tag{9.1}$$

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 (9.2), 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*.

$$\begin{aligned} Precision_{conf} &= \sum_{i=1}^{96} \frac{|AC_i| \cap |C_i|}{|C_i|} \\ Recall_{conf} &= \sum_{i=1}^{96} \frac{|AC_i| \cap |C_i|}{|AC_i|} \end{aligned} \tag{9.2}$$

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.

9.4.1. *Description of the resulting dataset.*

Table 5.2. 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
$\boldsymbol{Diff}_{012}$ 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

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[6]. Table 5.2 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 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.

Fig. 5.13 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

[6] `https://doi.org/10.5281/zenodo.7386968`

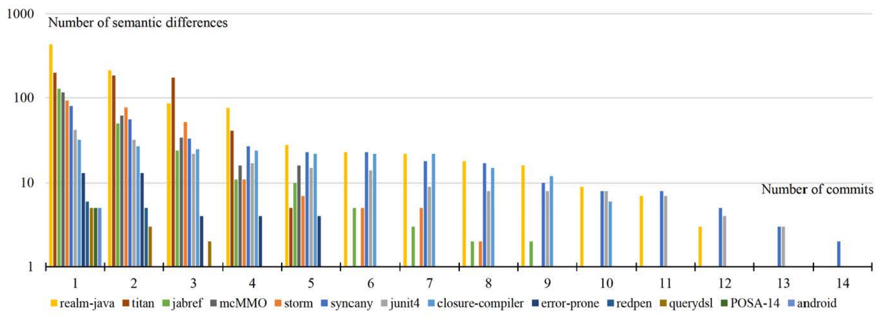


Figure 5.13. Number of semantic differences per commit

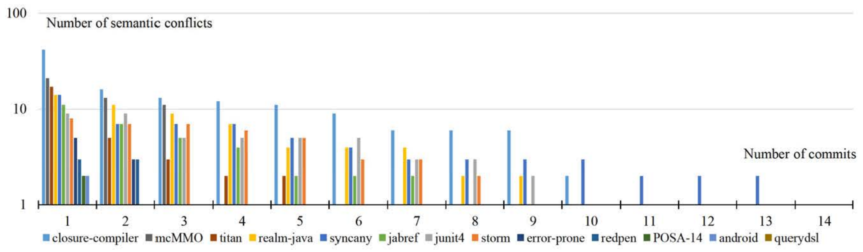


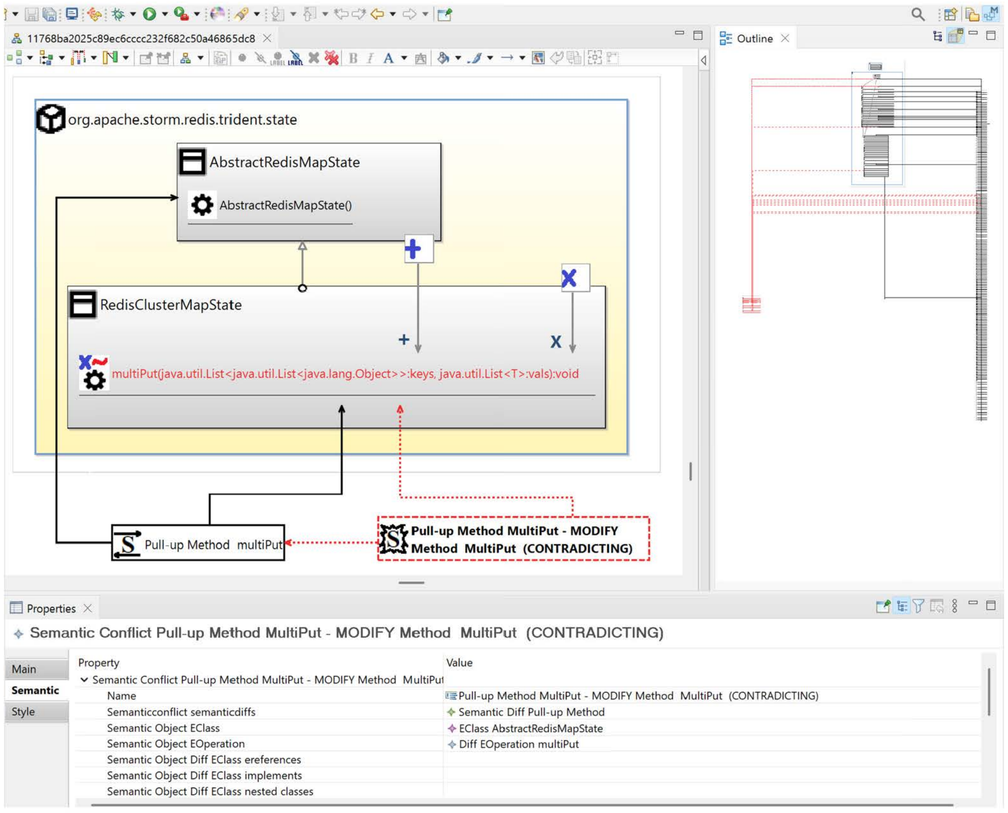
Figure 5.14. Number of semantic conflicts per commit

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 5.2 shows an average of 32 refactoring differences found in each of the 96 commits across all projects. The chart Fig. 5.13 characterizes the vast diversity of the dataset under study.

Fig. 5.14 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 (Fig. 5.13), 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.*

Fig. 5.15 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, DSM-Compare 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.


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

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 Fig. 5.15 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). Fig. 5.16 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 Fig. 5.16 is sorted in terms of the frequency of the semantic differences (refactorings). The most common refactoring differences found in the dataset (more than

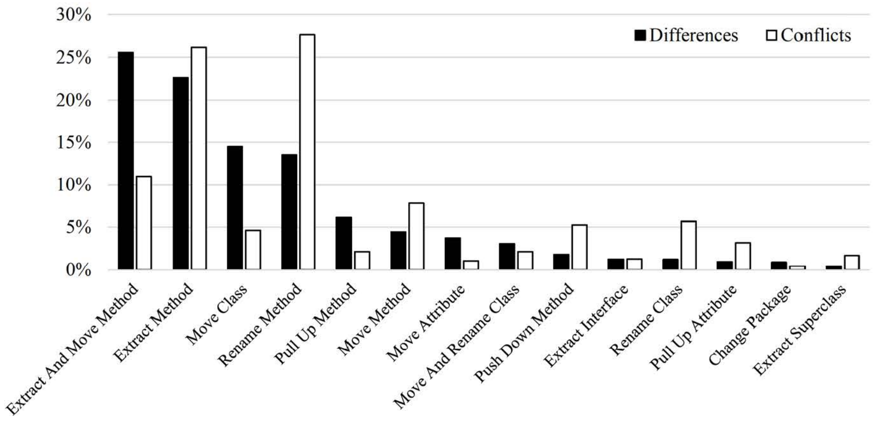


Figure 5.16. Frequency of refactoring types reported by DSMCompare compared to those involved in a conflict

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.

9.5. Effectiveness of DSMCompare

We now evaluate the results in terms of precision and recall. Fig. 5.17 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 (9.1) and (9.2) 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

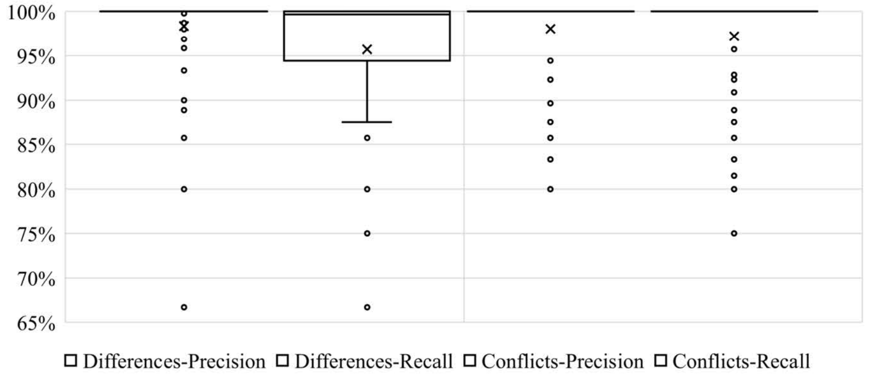


Figure 5.17. Precision and recall for detecting the semantic differences and conflicts

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.

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 Fig. 5.16, *Rename Method* is responsible for 13% of all differences but contributes to 28% of all the conflicts.

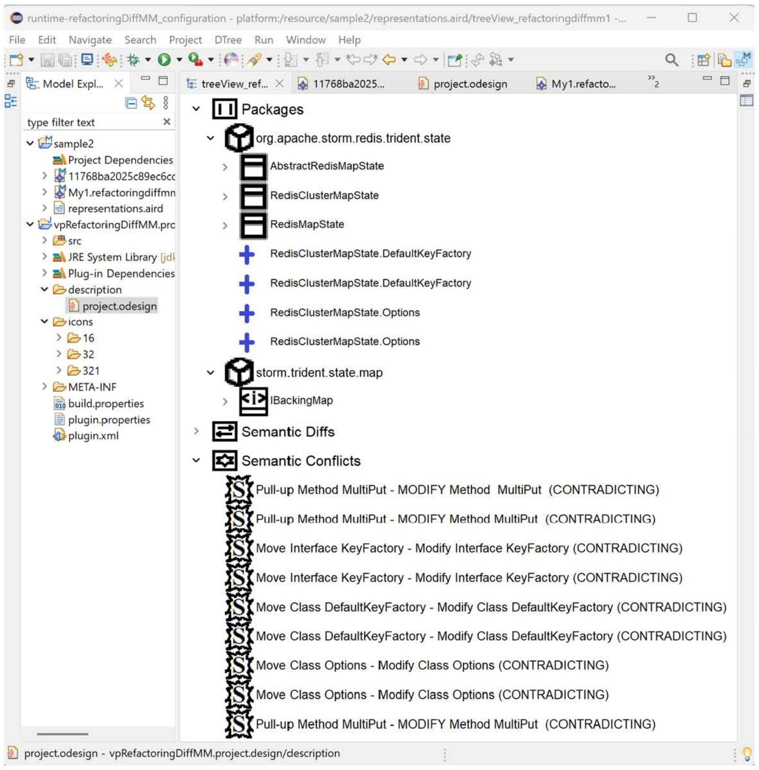
9.6.3. *Advantages of DSMCompare.*

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 syntax. Therefore, DSMCompare helps to understand and locate the exact problematic model elements conflicting with a semantic change. Our assumption is 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 Fig. 5.15, 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


runtime-refactoringDiffMM_configuration - platform:/resource/sample2/representations.aird/treeView_refactoringdiffmm1 -...
File Edit Navigate Search Project DTree Run Window Help
Model Expl...
treeView_ref...
11768ba2025...
project.odesign
My1.refacto...
type filter text

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 5.18. Visualization by Sirius, Tree-View presentation

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. Fig. 5.18 shows this alternative visualization of the same difference model presented in Fig. 5.15. 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[7] 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[8]. 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

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

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 Fig. 5.10). 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.

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 and 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 (2013a) post-processes atomic operations 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 (2011b). Rather, in the context of this paper, the term semantics pertains to editing semantics.

10.2. Conflict detection

Like DSMCompare, Brosch et al (2012b) 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, DSMCompare identifies all semantic differences, offering users a comprehensive overview of changes at a semantic level. This aids users in distinguishing the intended resolution when resolving conflicting semantic differences. Moreover, 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, since they do not provide users with a domain-specific mechanism to define new rules. Yet, they resemble the rules in our method.

Sharbaf and 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 and 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 abstract syntax 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 and 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, 2010c) 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[9] provides controlled access to all artifacts, simple configuration management, and a mechanism to prevent version conflicts in this manner. Obeo Designer[10] and CDO can integrate with EMFCompare to provide a generic model-based versioning service. Smart Model Versioning[11], 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

[9]`https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/` last accessed Jul 2022

[10]`https://www.obeodesigner.com/` last accessed Jul 2022

[11]`https://www.metacase.com/news/smart_model_versioning.html` last accessed Jul 2022

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 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.

Chapter 6

Domain-specific conflict resolution and model merge [1]

Manouchehr Zadahmad Jafarlou,
Eugene Syriani and Omar Alam
DIRO, Université de Montréal

This article represents the work I did for the third contribution of my thesis. The co-authors are my supervisors who contributed to the writing.

[1]The content of this chapter has been submitted to Journal of Systems and Software (JSS).

Résumé.

L'intégration de modèles élaborés grâce à un travail d'équipe collaboratif nécessite souvent des activités de résolution de conflits pour parvenir à une version successive cohérente. La résolution des conflits peut être effectuée manuellement ou automatiquement. Dans les deux approches, il est crucial de comprendre la sémantique des changements et des conflits. Malgré des recherches précieuses dans ce domaine, les systèmes de contrôle de version existants se concentrent principalement sur la syntaxe abstraite des modèles spécifiques à un domaine et sur les conflits à granularité fine. De plus, le mécanisme permettant de définir les règles de résolution des conflits n'est souvent pas convivial pour les experts du domaine. De plus, ils visualisent les concepts de résolution de conflits pour des modèles spécifiques à un domaine sur la base de leur syntaxe abstraite plutôt que de la syntaxe concrète du DSL.

Pour relever ces défis, nous avons précédemment introduit DSMCompare, un outil conçu pour comparer des modèles spécifiques à un domaine, détecter les différences et visualiser les conflits en utilisant la syntaxe concrète du DSL. Dans cet article, nous avons encore amélioré notre approche pour parvenir à une résolution des conflits avec un degré élevé d'automatisation. Nous avons également ajouté des fonctionnalités générées automatiquement pour aider les experts du domaine au cas où l'intervention de l'utilisateur serait requise. De plus, nous générons automatiquement des éditeurs spécifiques au domaine pour que les ingénieurs DSL puissent définir et maintenir des règles de résolution de conflits.

Pour évaluer DSMCompare, nous avons procédé à une rétro-ingénierie de l'historique des validations de plusieurs projets open source impliquant des modifications de refactorisation de code basées sur Java. Nos résultats démontrent que DSMCompare est efficace à la fois dans la résolution automatique des conflits avec une grande précision et dans la réduction du besoin d'interventions manuelles de l'utilisateur.

Abstract.

Integrating models evolved through collaborative teamwork often requires conflict resolution activities to achieve a consistent successive version. Conflict resolution can be performed manually or automatically. In both approaches, it is crucial to understand the semantics of changes and conflicts. Despite valuable research in this area, existing version control systems mainly focus on the abstract syntax of domain-specific models and fine-grained conflicts. Additionally, the mechanism for defining conflict resolution rules is often not user-friendly for domain experts. Furthermore, they visualize conflict resolution concepts for domain-specific models based on their abstract syntax rather than the concrete syntax of the DSL.

To address these challenges, we previously introduced DSMCompare, a tool designed to compare domain-specific models, detect differences, and visualize conflicts using the concrete syntax of the DSL. In this paper, we have further enhanced our approach to achieve conflict resolution with a high degree of automation. We have also added automatically generated features to assist domain experts in case user intervention is required. Additionally, we automatically generate domain-specific editors for DSL engineers to define and maintain conflict resolution rules.

To evaluate DSMCompare, we reverse-engineered the commit history of several open-source projects that involved Java-based code refactoring changes. Our results demonstrate that DSMCompare is effective in both automatic conflict resolution with high accuracy and reducing the need for manual user interventions.

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

1. Introduction

During collaborative development activities, developers often work simultaneously on shared models pulled from version control systems (VCS) to their respective branches (Zadahmad et al, 2022). These models are manipulated, and the changes made by developers are pushed back to integrate them with the modifications of other team members (David et al, 2021). However, this widely adopted practice suffers from contradicting conflicts during the merge process, which poses a significant challenge that needs to be addressed (Shen et al, 2019). Resolving these conflicts requires custom and elaborate solutions for each case (Brindescu et al, 2020), leading to delays in software development timelines.

While automatic conflict management is necessary, the existing practices are not yet mature enough to handle the complexities effectively (Sharbaf et al, 2022a). The approach to conflict management depends on how version control systems treat artifacts under version control. Popular tools like Subversion and Git use an unstructured approach that treats artifacts as textual documents (Mens, 2002). However, this approach does not consider the underlying meta-information of the artifacts, resulting in imprecise conflict reports that lack information about the type of conflicts and how to handle them. Moreover, these reports may include false positives due to the neglect of the artifact's meta-model (Sharbaf et al, 2022a).

Structured merging approaches, which focus on a single programming language like Java, are more precise but expensive to implement and often fail to represent program structures at an abstract level (Mahmoudi et al, 2019). Semi-structured merging treats artifacts partially as trees but sacrifices precision compared to structured merging. To overcome the limitations of tree-based approaches, graph-based structured merging approaches have been proposed, using conditional graph rewriting to manage the evolution of software artifacts (Mens, 2002). However, these approaches are more complex to implement due to the need for sophisticated program analysis and representation. Current works, such as *EMFCompare* (EMF Compare, accessed August 2023), focus on merging software models rather than programs. Nevertheless, they fall short to represent conflicts in a domain-specific manner, as the reports they provide use general object-oriented terminology that may not be understandable for DSL users (Zadahmad et al, 2022). Additionally, existing domain-specific works are limited to predefined modeling languages, cover only specific types of conflicts, or are challenging for DSL users to utilize in defining conflict resolution patterns (Sharbaf et al, 2022a). A recent survey on conflict management for model merging identified semantic conflict management and their visualization as crucial challenges to be addressed (Sharbaf et al, 2022b).

In this paper, we introduce a novel approach for domain-specific three-way model merging. Our approach offers a mechanism for specifying conflict resolution patterns and applying them to resolve merge conflicts. Additionally, we provide an algorithm for automatically resolving conflicts and tools for DSL users to navigate, resolve conflicts, and generate merged versions for any DSL, regardless of its abstract or graphical concrete syntax. To illustrate the feasibility of our approach, we have implemented it within DSMCompare (Zadahmad et al, 2022, 2019), a tool that already supports domain-specific conflict detection in three-way differencing. Furthermore, we assess the effectiveness of our domain-specific merging approach in comparison to model-based and line-based merging.

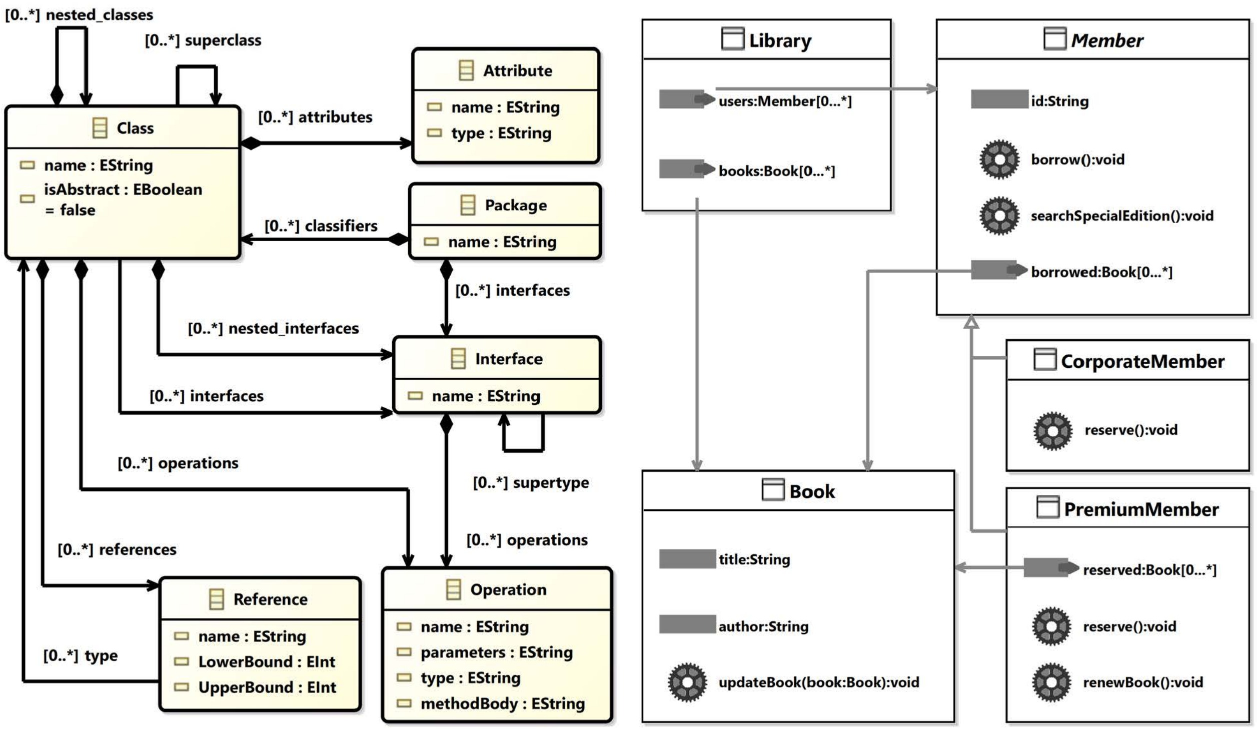
The structure of this paper is as follows. In Section 2.1, we the motivation behind our approach and introduce a running example. In Section 3, we focus on the conflict resolution model, while in Section 4 we explain the specifics of the domain-specific model merge algorithm. We explain automatic conflict resolution in Section 5. In Section 6, we describe how to tailor the concrete syntax of difference models and conflict resolution rules. In Section 7, we discuss the manual resolution process and in Section 8 we address the model merge process. We evaluate our approach in Section 9. Finally, we discuss related work in Section 10 and conclude in Section 11.

2. Motivation and conflict detection

We introduce a practical running example to motivate our approach and provide a brief overview of domain-specific conflict detection already supported in DSMCompare.

2.1. Running example

We illustrate three-way domain-specific model merging using the following running example. Consider a DSL to represent the structure of Java-based software supporting packages, classes, interfaces, attributes, references (attributes other than primitive data types), and operations. The relations considered between classes (and interfaces) are nesting and sub-typing (realization). Operations store the details of the signature as well as a string concatenating the entire body. A software engineer Charlie has defined the metamodel of Mini-Java implemented in Ecore. It is depicted in Fig. 6.1a. She has chosen a graphical concrete syntax inspired from UML class diagrams to represent Mini-Java models visually. Fig. 6.1b shows an example using the concrete syntax, that he has implemented in an Eclipse-based graphical modeling editor with Sirius (Sirius, 2023a). The legend outlines the mapping between the graphical symbols and the metamodel concepts. He has integrated EGit (Eclipse EGit,



(a) Metamodel of the Mini-Java DSL **(b)** The initial library model (*V0*)

Figure 6.1. The definition and an instance of Mini-Java

2023) in the editor to foster collaboration using a Git-based version control system within the Eclipse environment.

Consider the following collaboration scenario between two Mini-Java modelers, Alice and Bob. Suppose they are provided with the initial version (*V0*) of a library system shown in Fig. 6.1b. A library has books and members. The library can update the title and authors of a book. A member can borrow books and search for special editions. There are two special types of members who can benefit from additional features. Corporate and premium members can reserve books. The latter can also renew book loans.

Alice and Bob check out the *V0* model and modify it locally. Fig. 6.2 depicts their resulting versions. Alice removes the possibility to update books. She decides that searching is no longer restricted to special editions but to all books. To this end, she modifies the `searchSpecialEdition` operation accordingly and moves it to the `Book` class under the name `search`. She pulls up the `reserve` operation so that reserving books is now enabled to all members. Finally, she renames the `renewBook` operation to `bookRenew`. After making her changes, Alice commits them through EGit and requests a merge. EGit accepts the request and merges her model making it the new version *V1*.

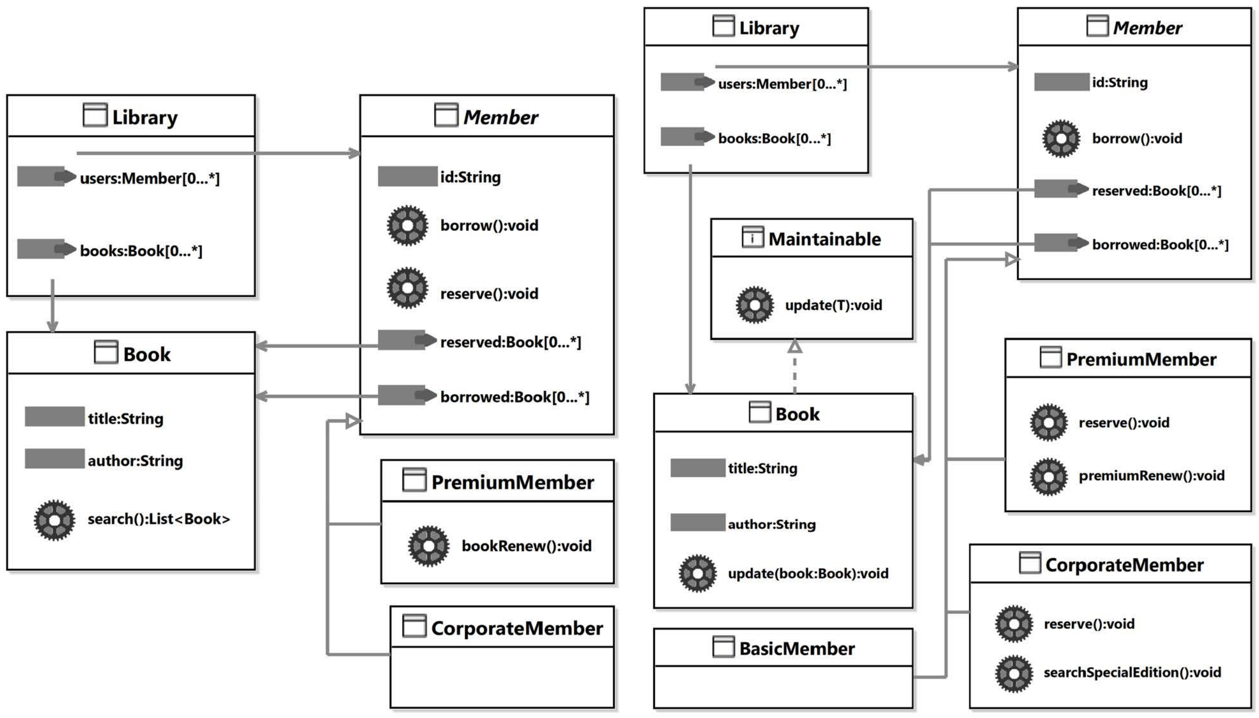


Figure 6.2. Alice (*V1*) and Bob's (*V2*) versions of the library system

As shown in Fig. 6.2b, Bob refactors the model differently. `Member` is now a generic abstract class that comprises a third type of basic members. Consequently, he pushes downs the `searchSpecialEdition` operation to be available to corporate members only. He creates a `Maintainable` interface to update books and renames the `updateBook` operation to conform to the new interface. Since renewing loans is specific to premium members, he renames the operation `renameBook` accordingly. After completing his work, Bob requests a merge. However, EGit rejects this merge request because Bob's model is incompatible with the latest version, Alice's V1. Being a line-based version control system, EGit reports the differences and conflicts at the XMI level. Consequently, comprehending, resolvling various conflict types, and merging models become challenging for Bob. To resolve this issue, Charlie aims to provide them with a domain-specific model comparison and merging tool to alleviate these challenges.

2.2. Detecting differences with DSMCompare

DSMCompare (Zadahmad et al, 2022) is a tool that compares and detects conflicts in domain-specific models. It sets itself apart from tools like EMFCompare (Brun and Pierantonio, 2008) and EMF DiffMerge (DiffMerge, 2023) by using a domain-specific approach. Unlike these tools, DSMCompare utilizes the same concrete syntax as the original DSL when presenting differences and conflicts. Furthermore, it can identify conflicts between coarse-grained differences (*i.e.,* semantic differences) and between semantic differences and fine-grained differences. As a result, it hides fine-grained differences or conflicts between fine-grained differences that are irrelevant to domain experts.

To seamlessly integrate DSMCompare into the Mini-Java editor, Charlie supplies as inputs to DSMCompare the Ecore metamodel of the DSL and `odesign` representation description, which defines the concrete syntax of Mini-Java in Sirius. DSMCompare then automatically generates a domain-specific model comparison tool designed specifically for comparing Mini-Java models, a.k.a. *MiniJavaCompare.*

MiniJavaCompare is tailored to the unique characteristics of Mini-Java models and allows for efficient and accurate model comparisons. Moreover, it provides two editors generated

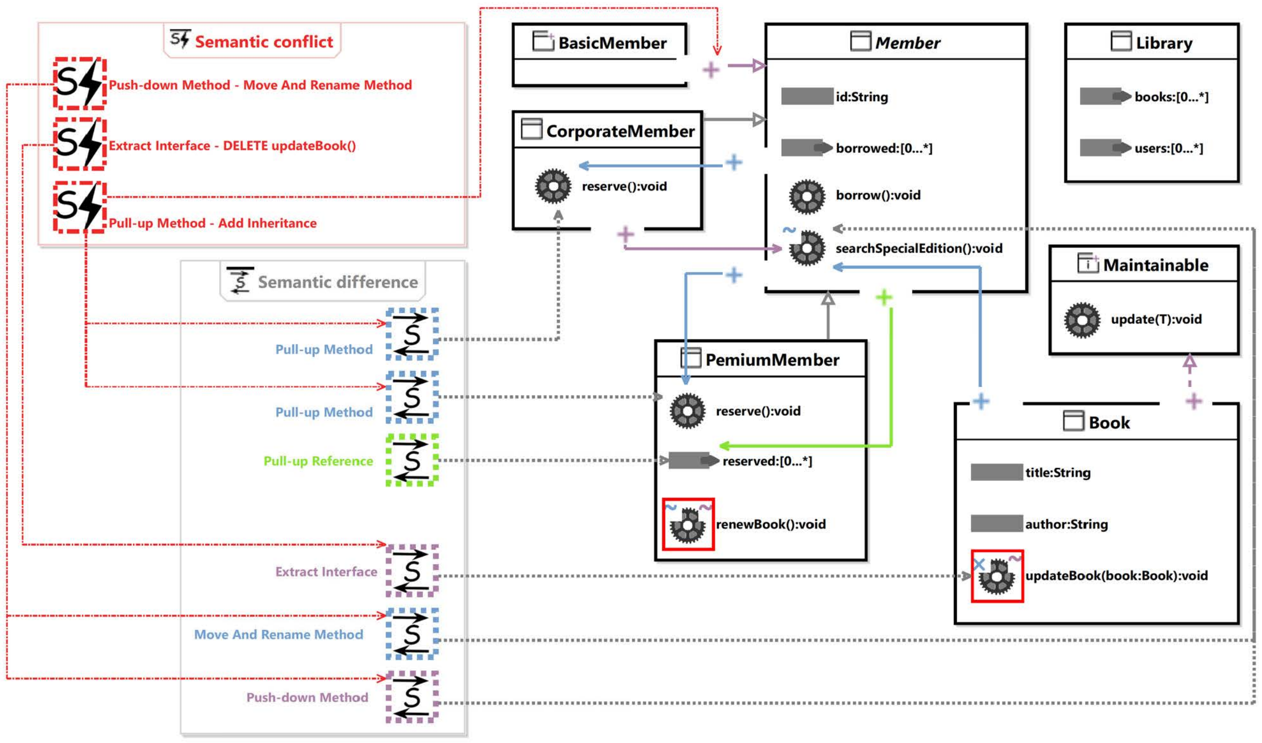


Figure 6.3. The three-way difference model Diff_{012} between V0, V1, and V2

based on the original DSL. The first editor, the *Diff*$_{012}$ model editor, facilitates the representation of differences, including semantic differences. Fig. 6.3 displays a portion of the editor for the running example. We notice that the concrete syntax of the difference model is similar to the original DSL, making it easier to comprehend the differences. MiniJavaCompare enhances clarity by using three default icons: $+$ for additions, $\times$ for deletions, and $\sim$ for modifications which are color-coded to distinguish the provenance of the changes (V1 in blue and V2 in purple in Fig. 6.3). Charlie, in her role as the DSL engineer, has the flexibility to customize these graphical representations according to the specific needs and preferences of the Mini-Java domain. Apart from visualizing the differences, the editor provides mechanisms to edit and resolve differences between *Mini-Java* models. Additionally, it catalogs semantic differences. A semantic difference semantically lifts fine-grained differences by aggregating them, hiding some of them, and assigning an interpretation to this encapsulation tailored to the semantic domain of the DSL. For Mini-Java, the semantic of changes can be interpreted as a refactoring pattern. For example, it reports that Alice (V1 in blue) has *pulled-up the method* `reserve`. To report these semantic differences, MiniJavaCompare must be aware of how to detect them.

The second editor available in MiniJavaCompare enables Charlie to define custom semantic differencing rules, *SDRules*, tailored specifically to the DSL. Fig. 6.4 displays the *Pull-up Method* SDRule. This rule is triggered when an operation is removed from a subclass and added to its superclass. If such a situation occurs in the *Diff*$_{012}$ model, the rule creates a semantic difference object with a descriptive name that reflects the intent of the semantic change. It also filters the deleted association from the subclass to the operation. Filtering is applied at the concrete syntax level to decrease the verbosity of the reported

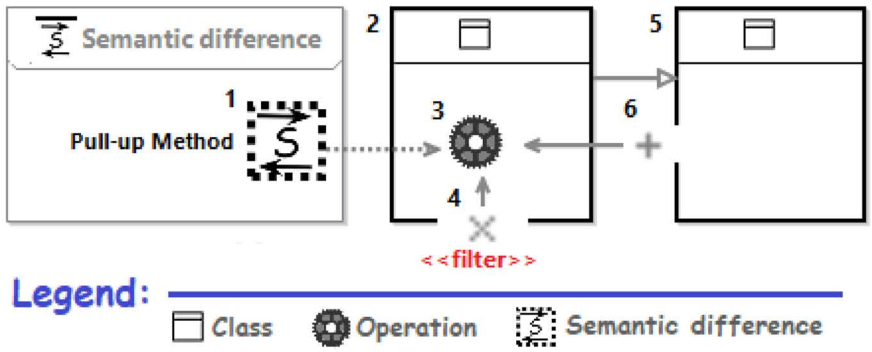


Figure 6.4. *Pull-up Method* semantic differencing rule

differences. Using this editor, Charlie defines 18 SDRules corresponding to object-oriented refactoring patterns within the *Mini-Java* domain. Fig. 6.3 shows that Alice has performed three semantic differences: *Pull-up Method* of operation `reserve` from `CorporateMember` and `PremiumMember`, and *Move and Rename Method* of operation `searchSpecialEdition`. MiniJavaCompare reports two semantic differences from Bob: *Push-down Method* of operation `searchSpecialEdition`, and *Extract Interface* of `Maintainable`. Both users also performed *Pull-up Reference* of `reserveds` from `PremiumMember` shown in green.

2.3. Detecting conflicts with DSMCompare

Recall the collaboration scenario when Bob's merge request failed because of conflicting changes. Upon requesting a merge, EGit utilizes the API to call upon MiniJavaCompare. It passes the versions the latest version (V1), the new version (V2), and their common ancestor (V0) of the library model as arguments. MiniJavaCompare, in turn, invokes the calculation of the domain-specific differences and conflicts. If there are conflicts that require user intervention, MiniJavaCompare reports the Diff_{012} model to Bob in a dedicated editor. Equivalent changes (not requiring user intervention) are shown in green.

Out of the six semantic differences shown in Fig. 6.3, MiniJavaCompare identifies three conflicts among them. One conflict shows that Bob's *Extract Interface* semantic difference contradicts Alice's deletion of the `update` operation. This is an example of a conflict between a semantic difference and a fine-grained difference (a.k.a. **semantic-fine conflict**). Another conflict shows that Alice's *Pull-up Method* contradicts Bob's addition of inheritance to from the `BasicMember` to the `Member` classes. This is also a semantic-fine conflict. The third conflict shows that Alice's *Push-down Method* contradicts Bob's *Move and Rename Method* of the `searchSpecialEdition` operation. This is a conflict between semantic differences (a.k.a. **semantic-semantic conflict**). Lastly, conflicts between fine-grained differences (a.k.a. **fine-fine conflict**) are surrounded with a red box in Fig. 6.3. One of the fine-fine conflicts indicates that Alice and Bob have renamed differently he `renewBook` operation. The other fine-fine conflict indicates that Alice has deleted the `update` operation whereas Bob has renamed it.

2.4. How DSMCompare detects differences and conflicts

We highlight the main internal workings of DSMCompare to explain how it computes three-ways differences and conflicting changes (see (Zadahmad et al, 2022) for a detailed explanation). DSMCompare expands the original DSL metamodel, referred to as `MM`, to incorporate semantic changes and conflicts in a three-way context. The core concept is to retain the inherent structural features and style of the original DSL, aligning with the fundamental essence of the DSL (Zadahmad et al, 2019). This process involves creating a new metamodel, labeled `DSDiffMM`, designed to capture specific data elements essential for three-way comparison. These elements encompass authorship details, changes in single- and multi-valued attributes and associations, detailed fine-grained and semantic differences, and various types of conflicts. Similarly, DSMCompare expands the original DSL concrete syntax, referred to as `CS`, to visualize the differences and conflicts in a graphical form. It creates a new concrete syntax definition `DSDiffCS` adapted to the `DSDiffMM` metamodel. This concrete syntax introduces a layering mechanism to view all differences, semantic differences, or conflicts, as well as layouting algorithm to group semantic differences and semantic conflicts.

To compute differences, DSMCompare relies initially on the EMFCompare API to report all fine-grained differences. With this information, it can construct the Diff_{012} model as an instance of `DSDiffMM`. It consolidates the changes from both versions in each model element. For example, each class of `DSDiffMM` holds a couple indicating the change operation performed in V1 and V2 (*e.g., MODIFY-DELETE*). Each attribute of `DSDiffMM` has a triplet holding the old value (from V0), the latest value (from V1), and the working value (from V2). To apply semantic differencing, the user defines SDRules graphically in a dedicated editor generated to specify the structural pattern that needs to be found in Diff_{012} model, the name of the semantic difference, and the fine-grained differences related to the semantic difference that can be hidden. SDRules are subsequently transformed into Henshin graph transformation rules (Strüber et al, 2017), facilitating their application to the Diff_{012} model. Since multiple SDRules may overlap on the same fine-grained differences, a strategic ordering of the application of the transformation rules maximizes the number of SDRules to be applied while minimizing the number of remaining fine-grained differences (Zadahmad et al, 2019).

Computing contradicting conflicts between fine-grained differences is trivial with DSMCompare. For each model element, it simply compares the changes from each version and, if they are different, they are marked as *CONTRADICTING*. To detect semantic-semantic and semantic-fine conflicts, DSMCompare uses a more advanced conflict and dependency analysis of the Henshin rules (Zadahmad et al, 2022). All these operations update the Diff_{012} model incrementally until the final model is presented to the user, like in Fig. 6.3.

2.5. Requirements for domain-specific model merging

To merge the changes into the successive version, Bob needs to resolve all the semantic and fine-grained conflicts. We list the following requirements to handle conflict resolution and model merging in the context of three-way model versioning for domain-specific models. The list stems from our experience in using line-based (de la Vega and Kolovos, 2022) and model-based (EMF Compare, accessed August 2023; Sharbaf and Zamani, 2020) version control systems, model merging techniques (Brosch et al, 2012c; Mansoor et al, 2015), research opportunities identified in a recent survey (Sharbaf et al, 2022b), and the feedback from users of an early version of DSMCompare.

Req 1. DSL users manage conflicts in an environment **familiar to their DSL**. It provides DSL users with a conflict management environment that aligns with their DSL's syntax and semantics, ensuring a familiar and intuitive experience. By doing so, we aim to enhance user productivity and reduce cognitive overhead, as users can leverage their domain-specific knowledge to efficiently resolve conflicts. Consequently, we anticipate that this tailored conflict resolution environment will lead to more accurate and context-aware conflict resolutions, ultimately improving the quality and reliability of the merged models.

Req 2. The tool manages **fine-grained and semantic conflicts**. It addresses conflicts at various levels of granularity within the models.

Req 3. The tool **resolves automatically trivial changes** performed by one or both versions. It reduces manual effort and expediting merging. This feature streamlines conflict resolution for straightforward modifications.

Req 4. The tool **resolves automatically semantic conflicts** when the appropriate resolution strategy is available. It enables automatic semantic conflict resolution, relying on predefined conflict resolution rules for efficient and accurate conflict handling.

Req 5. The tool **proposes semi-automatic resolutions** offering predefined choices to the user. This approach simplifies the decision-making process by providing structured choices, reducing the manual effort required.

Req 6. The tool supports **manual resolution** of unresolved conflicts to enable custom changes during the merge process. This capability allows users to intervene when necessary, ensuring flexibility in handling unique conflict scenarios during the merging process.

Req 7. Users **navigate between conflicts** and explore fine-grained conflicts underlying coarse-grained conflicts. It provides a comprehensive view of the conflict resolution process, empowering the user to navigate and understand the main intentions of changes at different levels of granularity. As a result, it enhances the user's control and confidence in the merging process.

Req 8. Users **undo** previous resolution decisions for each conflict and modify their resolution choices. This feature empowers users to align conflict resolutions more closely with evolving requirements or unanticipated changes in project goals.

Req 9. To handle conflicts in large models or when there is a significant amount of conflicts, users **save partial resolutions**, allowing them to resume the merge process at their convenience. It enables the user to effectively manage and prioritize their conflict resolution efforts.

Among the currently available tools compatible with the Eclipse Modeling Framework that we have evaluated (DSMDiff (Lin et al, 2007), DiffMerge (DiffMerge, 2023), E3MP (Sharbaf and Zamani, 2020), EMFCompare (EMF Compare, accessed August 2023), and DSMCompare), DSMCompare stands out as the only tool supporting domain-specific three-way comparisons that can be interactively manipulated through an API. Therefore, we choose DSMCompare to showcase our domain-specific approach for conflict resolution and merging. In particular, the remaining sections elaborate on how to address the abovementioned requirements.

3. Conflict resolution model

In this section, we present the data structure needed to resolve conflicts and merge the differences in a domain-specific way. To support the resolution of conflicts, we adapt the model used to track three-way differences and detect conflicts in DSMCompare according to the requirements in Section 2.5. First, we review the existing difference metamodel generated with DSMCompare, then we explain how we extend it to support conflict resolutions. Finally, we describe how we create a conflict resolution rule metamodel to automatically resolve conflicts involving semantic differences.

3.1. Domain-specific difference metamodel

As explained in Section 2.4, the $Diff_{012}$ model conforms to a metamodel `DSDiffMM` that captures the differences between three model versions tailored to the original DSL. Fig. 6.5 depicts a fragment of the `DSDiffMM` metamodel that DSMCompare generates for the Mini-Java example. Based on the original metamodel of Mini-Java, it adds the following elements to create `DSDiffMM`. To keep track of changes made by V1 and V2 to a class `C` with respect to the common ancestor, it adds a new `DiffC` class. For example, in Fig. 6.5, `DiffClass` inherits from `Class`.

Furthermore, it creates a `DiffAsc` class to denote changes associated with the original association `Asc`. This is the case with the `DiffClass_superclass` intermediate class representing differences related to the `superclass` association. The outgoing `targetCA` association

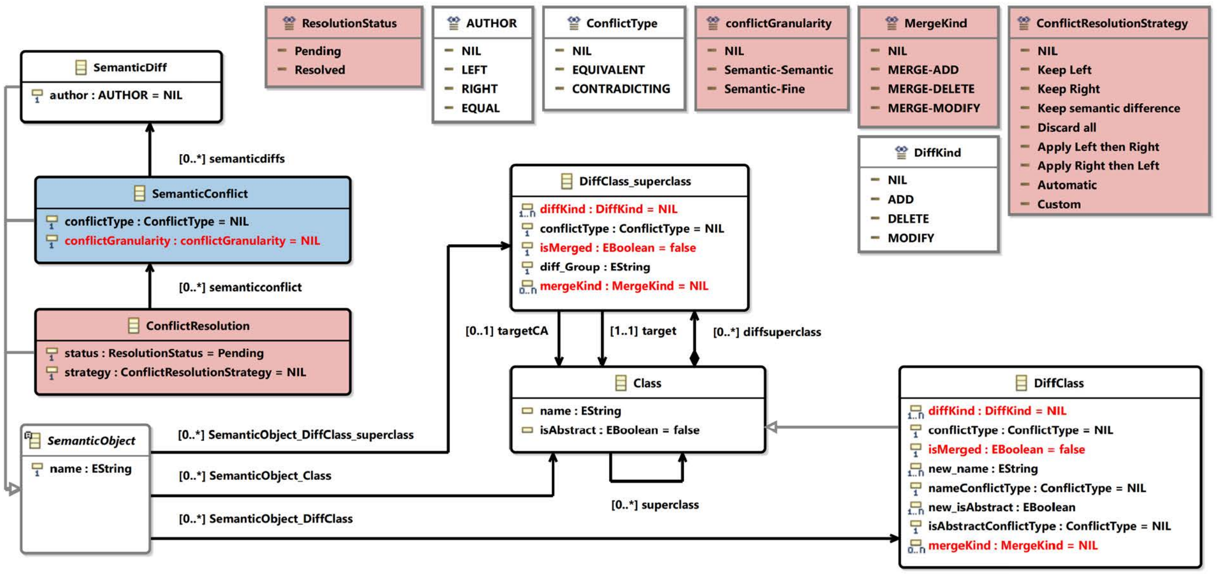

ResolutionStatus
Pending
Resolved
AUTHOR
NIL
LEFT
RIGHT
EQUAL
ConflictType
NIL
EQUIVALENT
CONTRADICTING
conflictGranularity
NIL
Semantic-Semantic
Semantic-Fine
MergeKind
NIL
MERGE-ADD
MERGE-DELETE
MERGE-MODIFY
ConflictResolutionStrategy
NIL
Keep Left
Keep Right
Keep semantic difference
Discard all
Apply Left then Right
Apply Right then Left
Automatic
Custom
DiffKind
NIL
ADD
DELETE
MODIFY
SemanticDiff
author : AUTHOR = NIL
[0..*] semanticdiffs
SemanticConflict
conflictType : ConflictType = NIL
conflictGranularity : conflictGranularity = NIL
[0..*] semanticconflict
ConflictResolution
status : ResolutionStatus = Pending
strategy : ConflictResolutionStrategy = NIL
SemanticObject
name : EString
[0..*] SemanticObject_DiffClass_superclass
[0..*] SemanticObject_Class
[0..*] SemanticObject_DiffClass
DiffClass_superclass
diffKind : DiffKind = NIL
conflictType : ConflictType = NIL
isMerged : EBoolean = false
diff_Group : EString
mergeKind : MergeKind = NIL
[0..1] targetCA
[1..1] target
[0..*] diffsuperclass
Class
name : EString
isAbstract : EBoolean = false
[0..*] superclass
DiffClass
diffKind : DiffKind = NIL
conflictType : ConflictType = NIL
isMerged : EBoolean = false
new_name : EString
nameConflictType : ConflictType = NIL
new_isAbstract : EBoolean
isAbstractConflictType : ConflictType = NIL
mergeKind : MergeKind = NIL

Figure 6.5. A fragment of the domain-specific difference metamodel `DSDiffMM` for Mini-Java. The adaptations to handle conflict resolutions are in blue (modified) and red (new).

denotes the original target in the common ancestor and the `target` association denotes the changed target of association.

To track changes in attribute values, DSMCompare creates a new array of size two typed with the original attribute type. For example, in `DiffClass`, the `isAbstract` attribute holds the value from CA and the `new_isAbstract` attribute holds the value from V1 in the first index and from V2 in the second index. Both `DiffC` and `DiffAsc` classes have an attribute of type `conflictType` to represent the conciliation of the changes from V1 and V2: *CONTRADICTING* means that the changes are conflicting and *EQUIVALENT* means the changes are compatible or similar.[2] Similarly, each attribute `A` is also associated with a new attribute `AConflictType` to track the change status of its value modified in V1 and V2. To store how each version has changed a model element, the `DSDiffMM` uses an array of size two called `DiffKind` indexed from left (V1) to right (V2) order. The kind of changes can be *ADD* if a new element is created in a version, *DELETE* if an element is removed from a version, or *MODIFY* if an attribute value or an association has changed.

To represent semantic differences, DSMCompare uses the `SemanticDiff` class. It employs an `Author` enumeration to keep track of the version accountable for each change. The values *LEFT* and *RIGHT* represent the two versions, while *EQUAL* indicates that both versions applied the same semantic change. The `SemanticDiff` class is also associated with all `DiffC` classes, thereby tracking which fine-grained differences are involved in a semantic difference. Finally, the `SemanticConflict` class tracks all contradicting and equivalent semantic-semantic and semantic-fine conflicts.

This data structure enables the Diff_{012} model to capture all differences and conflicts. However, it is not capable of handling the resolution of those conflicts and merging the differences.

3.2. Support for conflict resolution

There are various solutions to support automatic and manual conflict resolution due to concurrent changes (Sharbaf et al, 2022b). One viable approach is to isolate individual conflicting fine-grained differences from Diff_{012} and apply a dedicated mechanism for their

[2]In all the enumerations of the `DSDiffMM` metamodel, *NIL* is used as default value or when there is no change in one of the versions.

resolution and merge independently from the context (Westfechtel, 2014). However, particular scenarios necessitate a deeper contextual understanding beyond the scope of structurally related elements. Consequently, we opt to enhance the `DSDiffMM` metamodel by incorporating conflict resolution management concepts. This augmentation ensures the comprehensive consolidation of all pertinent information within a unified framework (Req 1).

3.2.1. *Conflict granularity*

Fig. 6.5 depicts the main concepts we have introduced in the `DSDiffMM` metamodel related to conflict resolution management. We augment the `DSDiffMM` with a conflict resolution object that can be associated with semantic conflicts and fine grained conflicts (`DiffC` and `DiffAsc` classes). This way, semantic-semantic, semantic-fine and fine-fine conflicts can be resolved individually (Req 2).

In order to facilitate effective conflict resolution and access for users, it becomes imperative to represent conflict resolution through an object. In response to this requirement, we've introduced the *ConflictResolution* class, designed to encapsulate a conflict resolution object. This class specializes from *SemanticObject* and inherits a name attribute, which serves the purpose of assigning a meaningful name to represent the semantic resolution.

Furthermore, the class incorporates attributes such as *status*, categorized under *ResolutionStatus*, and *strategy*, classified as *ConflictResolutionStrategy.* Additionally, a multi-valued association links it with *SemanticConflict*, effectively annotating the engaged semantic conflicts.

3.2.2. *Resolution strategies*

The primary requirement is to offer the adoption of appropriate strategies for resolving conflicting elements (Req 3)–(Req 6). Common strategies in version control systems and model-based tools, like EMFCompare, offer to choose between the changes from one version or the other, thereby prioritizing one version's intention over the other. This typically applies when there are conflicts at a fine-grained granularity. However, we also need to support scenarios involving semantic differences. For example, as shown in Fig. 6.1, the conflict *Push-down Method – Move And Rename Method* occurs between two semantic differences. In this case, favoring one of the semantic differences entails resolving all the conflicting fine-grained differences associated with it. Therefore, we introduce eight conflict resolution strategies (see the `ConflictResolutionStrategy` enumeration in Fig. 6.5).

Alongside preferring left or right versions, users can adjust the sequence of change application with, *e.g., Keep Left then Right* (Req 5). In some cases, users may want to fall back to the common ancestor version and *Discard all* choice is provided. In addition, DSMCompare supports advanced scenarios where frequent conflicts occur within a domain, and established resolutions exist. DSL engineers, like Charlie in our example, can specify these recurrent conflicts and define *Automatic* resolutions (Req 4). DSMCompare supports heuristic-based resolutions. For instance, in the case of a semantic-fine conflict, a semantic difference in one version contradicts a fine-grained difference in another. One heuristic is to interpret that a semantic change reflects a more carefully though intention, whereas a conflicting fine-grained change by the other user may have been a mistake, an oversight, or a corrective patch. Therefore, the user may decide to *Keep semantic difference*, in which case the semantic change prevails. Finally, some conflicts can only be resolved with a given context and the resolution is specific to a particular change in the given model. In such cases, the user needs to preform manual adjustment with a *Custom* resolution during merge time (Req 6).

3.2.3. *Tracking modifications to merge*

When a conflict is resolved automatically or manually (*automatic* and *custom* resolution), DSMCompare needs to keep track of the new changes in the $Diff_{012}$ model to perform the merge properly (Req 6). Therefore, every element of the `DSDiffMM` includes a `conflictType` that tracks if the conflict is equivalent or contradicting. In addition, every element of the `DSDiffMM` includes a `MergeKind` that tracks the the kind of change for the merge: *MERGE-ADD* for additions of `DiffC` and `DiffAsc` classes, *MERGE-DELETE* for their deletion, and *MERGE-MODIFY* for their modification as well as modifications of attributes and associations.

3.2.4. *Partial merge*

Dealing with a difference model containing an extensive amount of conflicting situation between versions is not uncommon, for example as a result of a major refactoring of the model. The model, such as Alice, may not have the time to resolve all conflicts during one contiguous session, and only resolve some conflicts and leave others for a later time. Therefore, DSMCompare must offer the ability to resume the merge process and, more specifically, to distinctly discern between resolved and pending conflicts (Req 9). The `ResolutionStatus` enables to track whether each individual conflict is *Resolved* or still *Pending*.

The *ResolutionStatus* addresses conflict resolutions associated with a conflict of different granularity. However, we also need to effectively annotate the merge status of fine-grained conflicting differences within the broader scope of conflict resolution. To address this requirement, for each diff class like *DiffC* or *DiffAsc*, we have added two new attributes. The attribute *isMerged* is included to indicate whether the changes have already been integrated into the final merge model (V_{012}).

3.3. Conflict resolution rule metamodel `CRRuleMM`

With the new extension, the `DSDiffMM` metamodel can now effectively represent conflict resolution concepts. However, in specific domains, conflicting patterns frequently arise, and proven solutions exist for their automatic detection and resolution. DSL users require domain-specific practices to specify and automatically resolve conflicts. Thus, we define a

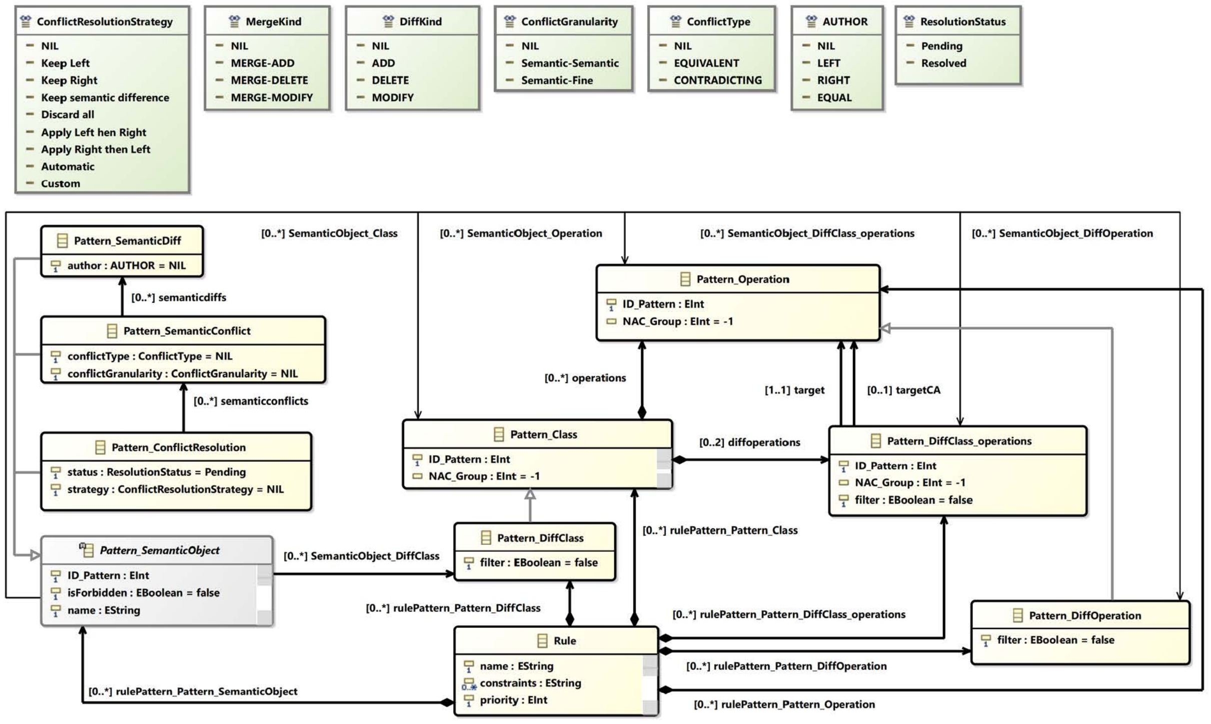


Figure 6.6. A fragment of Min-Java Conflict Rule Resolution metamodel (CRRuleMM)

conflict resolution rule as a predefined strategy governing the identification and resolution of conflicts. The details of conflict resolution rules are provided in section 5.2.1. This rule encompasses two components: the conflict specification pattern and the conflict resolution pattern. To fulfill this need, we automatically generate a DSL that includes the Conflict Resolution Rule metamodel (*CRRuleMM*) and its corresponding concrete syntax (*CRRuleCS*) from the `DSDiffMM` metamodel and its concrete syntax (`DSDiffCS`). We proceed as follows:

Fig. 6.6 illustrates essential components and relationships within the generated conflict resolution metamodel (`CRRuleMM`) for the *Mini-Java* DSL. This generation is an automatic process facilitated by `DSMCompare`. The purpose of this metamodel is to provide a structured framework for defining and applying conflict resolution rules.

The DSL engineer needs to easily recognize and associate specific conflict scenarios with their intended resolution patterns. Therefore, we need to align meta-elements in `DSDiffMM` with corresponding elements in `CRRuleMM`. Therefore, each class C in `DSDiffMM` corresponds to a pattern class *Pattern_C* in `CRRuleMM`. As depicted in figure 6.6, `Pattern_DiffClass_operations` generated shows the pattern class for the `operations` association and `Pattern_DiffClass` represents `Class`. To uniquely identify objects, we create for each *Pattern_C* additional attributes depicted by *ID_Pattern*.

To satisfy the user's need for a structured mechanism to define conflict resolution rules, we include in the `CRRuleMM` a root class called *Rule*. It contains a constraint to restrict the applicability of a conflict resolution pattern based on attribute value changes. It is also associated with all elements in `DSDiffMM` to encapsulate flexible and robust conflict specification and conflict resolution patterns.

Since the DSL engineer user may also want to filter some unnecessary fine-grained differences to decrease the verbosity. To address this, for each *Pattern_C* we introduced an attribute *filter* when set to true, results in hiding the applied fine-grained difference. Additionally, the rule could encompass negative application conditions (NACs) aimed at prohibiting the existence of certain elements (Ehrig et al, 2006). We introduce a `NAC_group` attribute to all classes prefixed with `Pattern_`. Much like certain transformation languages (Arendt et al, 2010), a set of rule elements assigned the same `NAC_group` value collectively forms a NAC. Several values of this attribute are employed to denote multiple NACs within the rule, each of which must remain unmatched for the rule to be deemed applicable.

Algorithm 7 The domain-specific model merge workflow

```
procedure ConflictResolution($Diff_{012}, V_{012}$)
    if $Diff_{012} = \emptyset$ then
        Choose $V_0, V_1, V_2$ //Three versions of a domain-specific model
        $Diff_{012} \leftarrow calculateDiff(V_0, V_1, V_2)$
    end if
    if $V_{012} = \emptyset$ then
        $V_0 \leftarrow Load("V0")$
        $V_{012} \leftarrow$ Clone($V_0$)
        $Diff_{012} \leftarrow$ MergeTrivialChanges($Diff_{012}, V_0$)
        $Diff_{012} \leftarrow ApplyPrio(\text{Diff}_{012}, \text{HCRRules})$ //Priority: pre-defined rules
        $Diff_{012} \leftarrow Resolve(\text{Diff}_{012})$ //Resolve and Merge
    end if
    $Conf \leftarrow GetNextConflict(\text{Diff}_{012})$
    while $Conf \neq \emptyset$ do
        Manually choose a strategy for a Conf (conflict) or do custom changes
        $Conf \leftarrow GetNextConflict(\text{Diff}_{012})$
    end while
    $Diff_{012} \leftarrow Resolve(\text{Diff}_{012})$
end procedure
```

The workflow presented in *Algorithm 7* outlines our domain-specific approach to conflict resolution and merging, catering to the needs of users in handling version differences and model updates. The process involves two primary inputs: *Diff*012, representing differences between versions, and *V*012, denoting the resulting merged version.

The workflow starts by checking if there are any differences in $Diff_{012}$. In case there are not, the algorithm guides the user through selecting three versions of their domain-specific model: V_0, V_1, and V_2, with *V*0 serving as a common ancestor. These versions are then compared using *calculateDiff*, which produces the *Diff*012 file detailing fine-grained and semantic differences, conflicts, and equivalences.

If *Diff*012 is not empty but *V*012 is, the algorithm assists the user in loading *V*0 and creating the initial *V*012. It further streamlines the process by merging non-conflicting and equivalent differences through the application of the *MergeTrivialChanges* method to *Diff*012. The specifics are elaborated in Section 5.1. Furthermore, it calls the *ApplyPrio* method to enforce the predefined (*i.e.*, priority) conflict resolution rules. The `ApplyPrio` function applies all the predetermined HCRRules, updating the *Diff*012 model. The details are explained in

Section 5.2. With the *Diff*012 model incorporating all the changes resulting from automatic conflict resolution, the *Resolve* method is employed to generate the subsequent version, *V*012

Upon integrating these resolutions, the algorithm enters a user-guided loop to address remaining conflicts. It systematically retrieves conflicts using *GetNextConflict*, allowing the user to choose strategies or introduce custom changes via *MiniJavaCompare* until either all conflicts are resolved or the user decides to stop. This iterative process empowers users to maintain control over conflict resolution, ensuring a tailored approach. Additionally, the user can restart the workflow when convenient without losing outcomes or repeating completed tasks. Details are available in Section 7.

Ultimately, after resolving all conflicts, the algorithm concludes with a final resolve and merge operation invoked through the *Resolve* method. Additional details are furnished in Section 8.

5. Automatic conflict resolution

Automatic resolution includes two parts: merging the trivial changes, Section 5.1, and resolving the conflicts with priorities, Section 5.2.

5.1. Initial merged version from trivial differences

To construct the V_{012} model from trivial changes, `DSMCompare` needs to integrate the trivial changes from both branches (*V1* and *V2*). The *Algorithm 8* outlines the procedure.

The *Merge Trivial Changes* algorithm iterates through the sorted list of differences. The sorting of differences ensures that changes are applied based on their dependencies, preventing inconsistencies and errors. For instance, changes to classes should be applied before changes to attributes, associations, or contained objects, preserving the intended relationships and preventing conflicts.

Then, it checks for conflicting contradictions and applies or removes differences accordingly. The algorithm ensures that only non-conflicting and equivalent changes are merged into the successive version, while contradicting conflicts are excluded. An equivalent change indicates similar change edits to the same element done by both branches.

The `IsRealConflicting` function examines conflicting dependencies, while the `IsRealContradicting` function confirms if a conflicting difference is contradictory. As

Algorithm 8 Merge Trivial Changes

```
1: procedure MergeTrivialChanges($Diff_{012}$, $V_{012}$)
2:    $sortedDifferences \leftarrow$ SortDifferences($Diff_{012}$)
3:    for each $diff$ in $sortedDifferences$ do
4:       if not IsRealConflicting($diff$) then
5:          if $diff.diffKind = ADD$ or $diff.diffKind = MODIFY$ then
6:             $diff.applyTo(V_{012})$ //Merge
7:          else if $diff.diffKind = DELETE$ then
8:             $Remove(diff)$
9:          end if
10:      end if
11:   end for
12: end procedure
13: function IsRealConflicting($diff$)
14:    for each $dependency$ in $diff.getDependency()$ do
15:       if $dependency.conflictKind = CONFLICTING$ then
16:          if $IsRealContradicting(dependency)$ then
17:             return true
18:          end if
19:       end if
20:    end for
21:    return false
22: end function
23: function IsRealContradicting($conflictingDiff$)
24:    if $conflictingDiff.diffKind = MODIFY$ and $conflictingDiff.getConflict()$ != null and
       $conflictingDiff.getConflict().conflictKind = CONTRADICTING$ then
25:       return true
26:    end if
27:    return false
28: end function
29: function SortDifferences($differences$)
30:    $sortedDifferences \leftarrow Copy(differences)$
31:    $sortedDifferences.sort(Comparator(\lambda diff1, diff2$:
32:             return $1$ if $HasDependency(diff1, diff2)$ else
33:                   if $HasDependency(diff2, diff1)$ else $0$))
34:    return $sortedDifferences$
35: end function
36: function HasDependency($sourceDiff$, $targetDiff$)
37:    return $sourceDiff.getDependancy().contains(targetDiff)$
38: end function
```

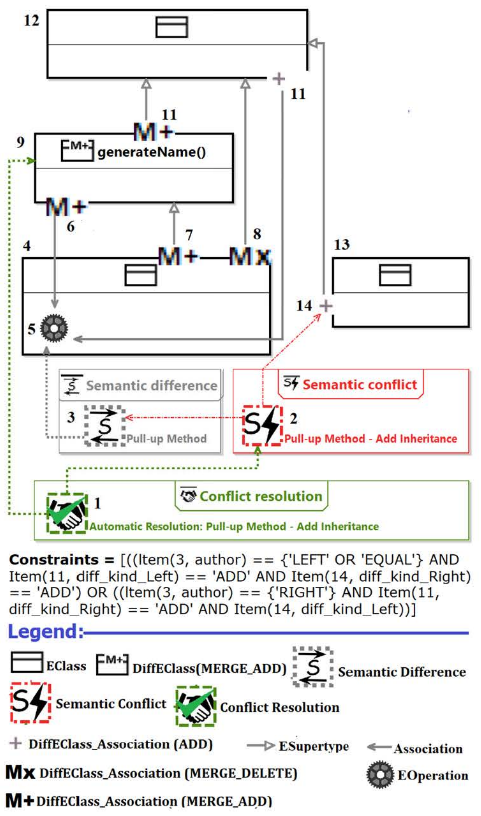
shown in figure 6.1, only Bob's introduction of the new class *BasicMember* and interface *Maintainable* is not contradictory. All other changes have syntactic or semantic contradictions. Consequently, at this stage, only these specific changes are eligible for merging.

5.2. Manage priorities

In Section 3, we described how DSMCompare automatically generates *CRRuleMM*. In this section, we elaborate on how DSMCompare users can utilize the DSL generated from *CRRuleMM* to formulate conflict resolution rules and apply them for conflict resolution. Initially, we establish a clear definition of conflict resolution. Subsequently, we delineate the specification of automatic resolution rules within the context of three-way domain-specific model differencing. Additionally, we expound upon the process of Synthesis of automatic resolution rules. Furthermore, we explain how DSMCompare supports automatic resolution for different kind of semantic conflicts.

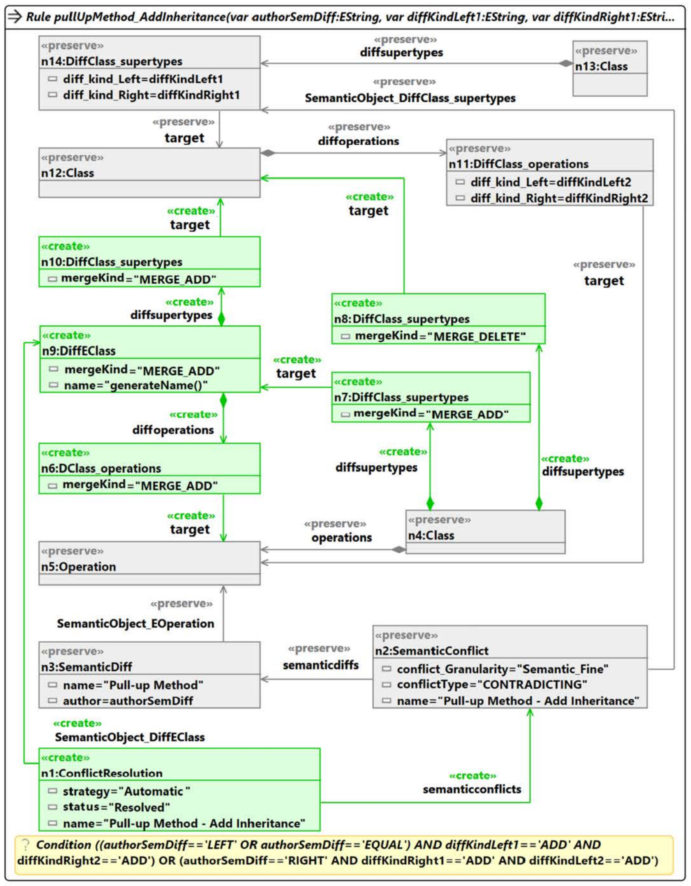
5.2.1. *Definition of conflict resolution*

In DSMCompare, a conflict resolution is defined as



Constraints = [((Item(3, author) == {'LEFT' OR 'EQUAL'} AND Item(11, diff_kind_Left) == 'ADD' AND Item(14, diff_kind_Right) == 'ADD') OR ((Item(3, author) == {'RIGHT'} AND Item(11, diff_kind_Right) == 'ADD' AND Item(14, diff_kind_Left))]

(a) The "Pull-up Method - Add inheritance" conflict resolution rule



Rule pullUpMethod_AddInheritance(var authorSemDiff:EString, var diffKindLeft1:EString, var diffKindRight1:EStri...
«preserve» n14:DiffClass_supertypes
diff_kind_Left=diffKindLeft1
diff_kind_Right=diffKindRight1
«preserve» diffsupertypes
«preserve» n13:Class
«preserve» SemanticObject_DiffClass_supertypes
«preserve» target
«preserve» diffoperations
«preserve» n12:Class
«preserve» n11:DiffClass_operations
diff_kind_Left=diffKindLeft2
diff_kind_Right=diffKindRight2
«create» target
«create» target
«create» n10:DiffClass_supertypes
mergeKind="MERGE_ADD"
«preserve» target
«create» diffsupertypes
«create» n8:DiffClass_supertypes
mergeKind="MERGE_DELETE"
«create» n9:DiffEClass
mergeKind="MERGE_ADD"
name="generateName()"
«create» target
«create» n7:DiffClass_supertypes
mergeKind="MERGE_ADD"
«create» diffoperations
«create» n6:DClass_operations
mergeKind="MERGE_ADD"
«create» diffsupertypes
«create» diffsupertypes
«create» target
«preserve» operations
«preserve» n4:Class
«preserve» n5:Operation
«preserve» SemanticObject_EOperation
«preserve» n3:SemanticDiff
name="Pull-up Method"
author=authorSemDiff
«preserve» semanticdiffs
«preserve» n2:SemanticConflict
conflict_Granularity="Semantic_Fine"
conflictType="CONTRADICTING"
name="Pull-up Method - Add Inheritance"
«create» SemanticObject_DiffEClass
«create» n1:ConflictResolution
strategy="Automatic "
status="Resolved"
name="Pull-up Method - Add Inheritance"
«create» semanticconflicts

Condition ((authorSemDiff=='LEFT' OR authorSemDiff=='EQUAL') AND diffKindLeft1=='ADD' AND diffKindRight2=='ADD') OR (authorSemDiff=='RIGHT' AND diffKindRight1=='ADD' AND diffKindLeft2=='ADD')

(b) The "Pull-up Method and Add inheritance" rule in Henshin

Figure 6.7. A conflict resolution rule transformed into a Henshin graph transformation rule

$SD = \langle Meaning, Constraints, Context, DiffKind \rangle$.

Constraints represent a collection of constraints applied to a list of elements within the $Diff_{012}$ model. For instance, as illustrated in Fig. 6.7a, these constraints are visualized as a graph, accompanied by a condition that the graph pattern must adhere to.

DiffKind consists of a list of fine-grained differences present in *Constraints*. This list is used to specify the fine-grained differences that are created, removed, or modified during the application of a conflict resolution pattern.

Context denotes a list of elements within *Constraints*. These elements pertain to the conflict resolution after the lifting process, providing a contextual backdrop to the interpretation of the conflict resolution's meaning.

Meaning is a string expressed in the vocabulary of the conflict resolution of the DSL and the conflicts in the editing semantics of the *Constraints*. It is the interpretation of solving the semantic conflicts.

- Semantic conflict pattern (Conflict Specification)

Conflicts involving semantic differences can encompass multiple fine-grained elements that contradict each other (Zadahmad et al, 2022). A semantic conflict can arise between two semantic differences or between a semantic difference and a fine-grained difference. Moreover, we also categorize cases as semantic conflicts in which a semantic variation doesn't include any fine-grained contradictory difference but still violates the domain's semantics. For instance, as shown in Fig. 6.1, *Pull-up Method - Add Inheritance* represents a semantic-fine conflict, whereas *Push-down Method - Move And Rename Method* is a semantic-semantic conflict. Additionally, in this example, the latter conflict cannot be identified through syntax.

We define conflicts using a graph composed of elements from a `DSDiffMM` metamodel. These elements fulfill a combination of relations between elements and constraints, where each constraint outlines the satisfaction of a precise condition among the elements of the conflict pattern. The elements can cover various types, including fine-grained differences, semantic differences, and semantic conflicts. The presence of the defined specification within the `DSDiffMM` metamodel designates a semantic conflict pattern.

The aggregation of fine-grained differences, semantic differences, fine-grained conflicts, and semantic conflicts is stated in the pattern of the conflict resolution rule as a set of constraints (c.f. the Constraints component of the definition of SD).

For instance, Fig. 6.7a illustrates the *Pull-up Method - Add inheritance* conflict resolution rule *CRRule*. In this example, the conflicting situation can be described as follows: In one version, a method (M) is pulled up from subclasses to the parent class. In another version, an inheritance relationship to the parent class is introduced from a separate class (D), which does not initially contains the method (M) (Brosch et al, 2010b).

As illustrated in Fig. 6.7a, we specify the semantic conflict pattern of the *CRRule* as follows: A semantic conflict should have the name *Pull-up Method - Add inheritance* (labeled **2**). The semantic conflict must have an outgoing association to a semantic difference with the name *Pull-up Method* (labeled **3**). It must also have another association with an instance of *DiffClass_supertypes* (labeled **14**) which represents the contradictory *Add inheritance* part. The semantic difference must be associated with the pulled method (labeled **5**) and it should be contained in the subclass (labeled **4**). The *DiffClass_supertypes* must be associated with the superclass (labeled **12**).

The superclass must contain an instance of *DiffClass_operations* (labeled **11**) associated with the pulled method. As depicted in the condition, both *DiffClass_supertypes* and *Diff-Class_operations* should be added from different versions based on the value of the author in the semantic difference.

- Conflict resolution pattern

The conflict resolution pattern forms a graph (Cicchetti et al, 2008b; Sharbaf and Zamani, 2020). The root element within the graph represents a conflict resolution with a meaningful name and a conflict resolution strategy. Typically, the name conveys the semantic pattern and the solution provided for it. The resolution strategy specifies the decision made either by the DSL user or automatically by DSMCompare (based on the resolution pattern) to resolve the conflict. The root element generally links to one or more semantic conflicts and other related elements in the `DSDiffMM` metamodel.

We describe the conflict resolution aspect of the *Pull-up Method - Add Inheritance* rule as follows: To resolve the conflict, we introduce an intermediary class (I) that inherits from the parent class. The method (M) is then removed from the parent class and added to (I). Subclasses that initially had (M) will now inherit from (I) instead of the parent class, and their direct inheritance from the parent class is eliminated. This guarantees that the (M) method remains inaccessible to the (D), thus preventing any incorrect usage of the method in the merged version. Additionally, it ensures the proper inheritance for (D).

Please note that, as illustrated in Fig. 6.7a, *MiniJavaCompare* employs distinct symbols (**M+**, **M×**, and **M~**) to differentiate between additions, deletions, and modifications, respectively, throughout the merging process.

We also specify the semantic conflict pattern of the *CRRule* as follows: The conflict resolution object at the root should bear the name *Pull-up Method - Add inheritance* (labeled **1**), with the resolution strategy set to *"Automatic"* implying it will be executed automatically. The root element connects to a single semantic conflict in the graph (labeled **2**). Additionally, the root element connects to a newly created class added to the $Diff_{012}$ model (labeled **9**).

This class serves as an intermediary between the superclass and subclass, annotated with *MERGE-ADD*. The class's name will be automatically generated using the *generateName()* method, relying on the name of the subclass from which the method will be pulled up. The new class must encompass an instance of *DiffClass_supertypes* (labeled **11**), annotated with *MERGE-ADD* to enable specialization from the superclass. It must also include an instance of *DiffClass_operations* (labeled **6**), annotated with *MERGE-ADD* to incorporate the pulled method.

Furthermore, a context is essential to rectify the inheritance relationships tied to the subclass. Consequently, the conflict resolution introduces an instance of *DiffClass_supertypes* (labeled **7**), annotated with *MERGE-ADD*, to the subclass, permitting specialization from the new class. Similarly, another instance of *DiffClass_supertypes* (labeled **8**), annotated with *MERGE-DELETE*, is added to the subclass, allowing the removal of inheritance from the superclass as it now inherits from the new class.

5.2.2. *Synthesis of automatic resolution rules*

To apply the *CRRules*, we convert each *CRRule* into a Henshin graph transformation rule that maintains semantic equivalence, known as an *HCRRule*. These generated *HCRRules* are then executed on the $Diff_{012}$ model to implement conflict resolutions. Fig. 6.7a demonstrates the corresponding Henshin rule, *HCRRule* for the *Pull-up Method - Add Inheritance* rule.

We now outline the transformation processes to generate a Henshin graph transformation rule *CRRule* from an *HCRRule.* As an example, we use the *Pull-up Method and Add inheritance* rule depicted in Fig. 6.7.

(1) Create an HCRRule with the same name as the CRRule.

(2) Create a node in HCRRule with the action *preserve* for every pattern object in CRRule that is not a *ConflictResolution* object or any Diff object with mergeKind assigned to merge-related value (such as *MERGE-ADD*) (*e.g.,* node `n4` in the example).

(3) Create a node with the action *create* in HCRRule for each *ConflictResolution* object or any Diff object with mergeKind assigned to merge-related value (such as *MERGE-ADD*) in CRRule (*e.g.,* node `n1`).

(4) Create a node with the action *delete* in HCRRule for every pattern object with *filter* set to true in CRRule.

(5) Create a node with the action *forbid* in HCRRule for every pattern object with a *NAC_group* set in CRRule. Set the *forbid* identifier to the value of the *NAC_group*.

(6) Create a condition in HCRRule for each condition defined in CRRule.

(7) Create an edge with action *create* in HCRRule for each association adjacent to a *ConflictResolution* node in CRRule or merge-related Diff objects (*e.g.,* `DiffClass_esupertypes` between nodes `n10` and `n12`).

(8) Create an edge with action *delete* in HCRRule for each association adjacent to a pattern object with *filter* attribute set to a true in CRRule .

(9) Create an edge with action *forbid* in HCRRule for each association adjacent to a pattern object with *NAC_group* attribute set to a value in CRRule.

(10) Create an edge with action *preserve* in HCRRule for each association adjacent to a pattern object with *NAC_group* and *filter* attributes not set to a value in CRRule.

5.2.3. *Support for different kind of semantic conflicts*

We describe the support of DSMCompare for different kinds of conflicts using the running example.

In the case of a conflicting commit, *MiniJavaCompare* opens the *Diff*$_{012}$ model using an automatically generated editor, as shown in Figure 6.8. This editor not only displays the

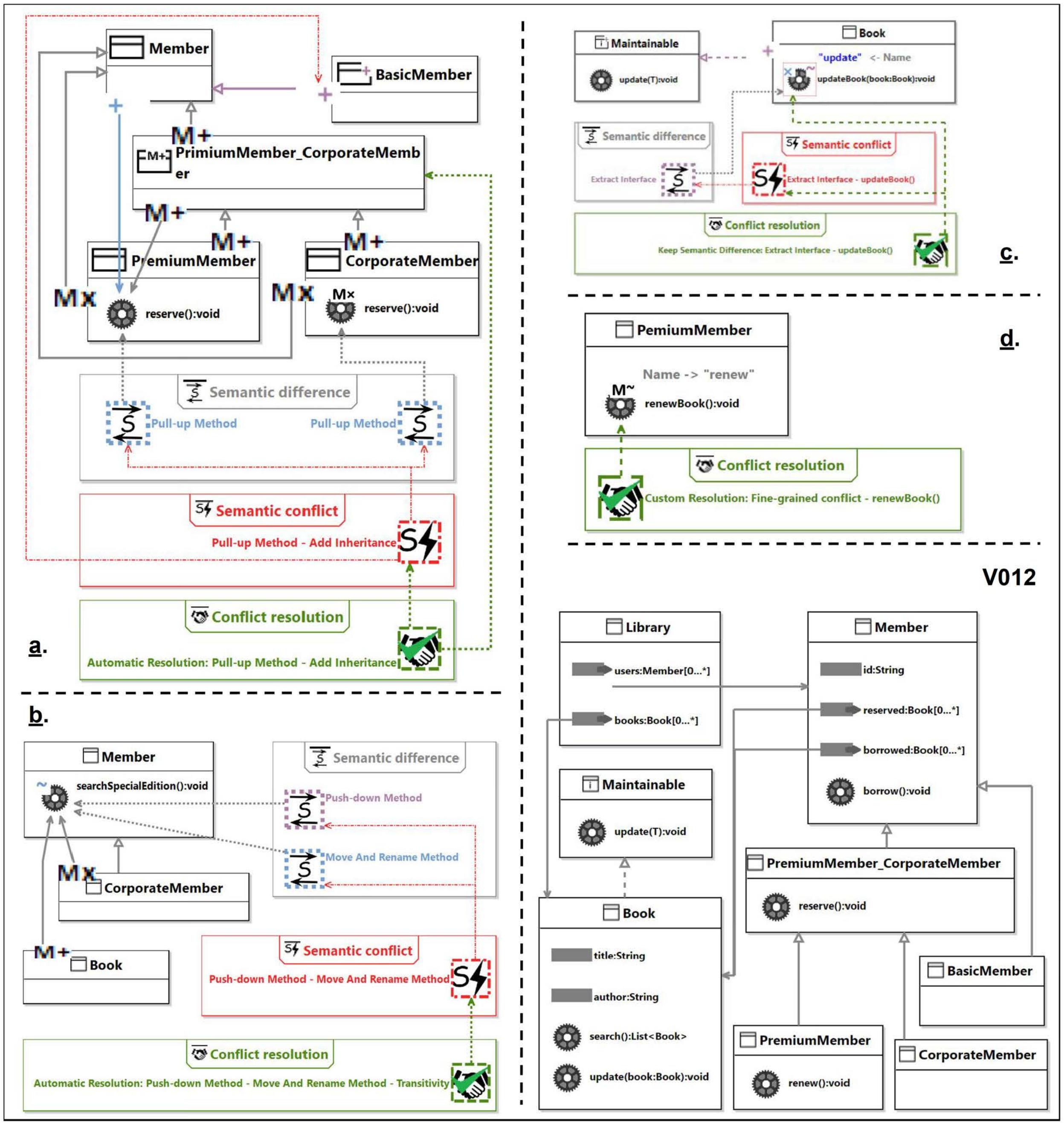


Figure 6.8. Domain-specific three-way Merge with DSMCompare - conflict resolution

content of the *Diff*$_{012}$ model but also provides tools and commands to facilitate the conflict resolution and merging process.

MiniJavaCompare automatically resolved the second and third conflicts by applying pre-defined conflict resolution patterns and updating the V_{012} model accordingly. It suggested a pre-defined *Keep Semantic Difference* approach for the first conflict (subfigure c) and generated a resolution pointer to a fine-grained conflict not associated with any semantic conflict (subfigure d).

To address the second conflict (subfigure a), *MiniJavaCompare* automatically applied the *Pull-up Method - Add Inheritance CRRule*, as depicted in Figure 6.7a. As a result, it introduced an intermediary class named *PremiumMember_CorporatedMember* between the superclass *Member*, and the subclasses *PremiumMember* and *CorporatedMember*. The name of this intermediary class was generated using the *generateName* method. Additionally, it moved the method *reserved* to *PremiumMember_CorporatedMember*.

For the resolution of the third conflict (subfigure b), *MiniJavaCompare* once again automatically applied another pre-defined *CRRule* called *Move and Rename Method / Push Down Method Transitivity*, inspired from (Ellis et al, 2023) and defined by Charlie. This conflict resolution recognizes the situation as a transitive relationship between the move and rename operation and the push-down operation. It ensures that the method is correctly moved to the target class while retaining its renamed name. In this example, *MiniJavaCompare* applied the *CRRule*, ensuring that *searchSpecialEdition* is properly moved to *Book* and retains its renamed name as *search* for the method.

As depicted in Figure 6.8, Bob accepts *MiniJavaCompare's* suggestion to *Keep Semantic Difference* for the first conflict (subfigure c). Consequently, the deletion of the method from the other branch is rejected, and in the subsequent version, the method *updateBook* is renamed to *update*, where it implements the method *update(<T>):void* in the `Maintainable` interface.

The last conflict (subfigure d) represents a fine-fine conflict where the name of the method *renewBook* changed to different values in both versions. Bob customarily resolves this conflict by renaming the method to the general name *renew*. After resolving all conflicts, he commits and pushes the changes.

MiniJavaCompare successfully resolves the remaining two conflicts and updates the V_{012} model. The merged model of V_{012} is illustrated in Figure 6.1. As evident in Figure 6.8 (V012), all decisions are incorporated into the final model.

It's worth emphasizing that the rules created by Charlie deal with the initial range of conflicts. However, *MiniJavaCompare* allows Charlie to incorporate new conflict resolution rules when necessary.

6. Tailoring the concrete syntax of difference models and conflict resolution rules

To generate graphical modeling workbench, DSMCompare employs Sirius (Sirius, 2023b) that is a well-known framework renowned for its capability to create graphical modeling environments within Eclipse. Sirius uses the viewpoint specification model (`odesign`) to define concrete syntax and establish connections between graphical representations and elements in the Metamodel (*MM*).

DSL engineers contribute their DSL's Concrete Syntax (*CS*) using Sirius, which DSM-Compare leverages to automatically generate the other editors. Across these editors, DSM-Compare maintains the core characteristics of the DSL by generating a default domain-specific difference Concrete Syntax (*DSDiffCS*), which inherits styles from *CS*. Additionally, (*CRRuleCS*) extends from *DSDiffCS*, ensuring continuity and consistency throughout the editing experience.

The `odesign` provides different meta-classes to visualize DSL elements. DSMCompare intelligently selects the appropriate meta-class from `odesign` based on the structural features of the elements. Charlie can further customize these representations. For instance, within the *MiniJava* metamodel, the `Class` element contains other meta-classes such as `Attribute`. *MiniJavaCompare* automatically employs a `ContainerNodeMapping` to represent the `Class` element and allows Charlie to provide a suitable icon for it.

However, since `Attribute` lacks contained elements, *MiniJavaCompare* automatically utilizes `NodeMapping` from `odesign` to represent it. The default style theme, encompassing properties like size and color, can be adjusted by Charlie. In terms of compositions, *MiniJavaCompare* employs a `BorderedNode_Mapping` within a `NodeMapping` to display a target class within the container class. For example, *DiffClass_supertypes* is generated using a `BorderedNode_Mapping`, with different icons and themes assigned based on conditions.

MiniJavaCompare harnesses the Acceleo Query Language (AQL) to automatically define distinct conditional graphic representations and themes. These queries are shaped using predefined themes and structural features of meta-classes. For instance, it generates a query to dynamically alter the color and icon of a *ContainerNodeMapping* representing a *ConflictResolution* to green when *status* changes to `Resolved`.

Automation extends to the creation of palettes that provides tools for DSL users to instantiate DSL elements. Additionally, different properties pages are generated, offering users a platform to precisely tailor element appearance and behavior. *MiniJavaCompare* further organizes semantic differences, conflicts, and resolutions into distinct containers, applying specific group themes and actions. This architecture enables users to conveniently manage elements within the group, such as hiding or showing all of them.

6.1. Automated Synthesis of Concrete Syntax

We provide some details on how DSMCompare enables the synthesis of Concrete Syntaxes. *DSDiffCS* includes styles for all the meta-classes of *CS*. For each class `C`, we reuse the same styling for the corresponding `Diff_C` class. For instance, for `Class` and `DiffClass` in *DSDiffCS*, and `Pattern_Class` and `Pattern_DiffClass` in *CRRuleCS*, we use the same styling of `Class` in *CS*. However, the `Diff_` classes have extended styling since they carry differencing properties. Therefore, *MiniJavaCompare* creates conditional styling, especially based on *diffKind*, *mergeKind*, and `conflictType`. As a result, for each `NodeMapping`, `ContainerNodeMapping`, and `BorderedNode_Mapping`, we generate 21 conditional styles. For instance, an AQL query such as *"aql:(self.diffKind_Left.toString() = 'MODIFY' or self.diffKind_Right.toString() = 'MODIFY') and self.conflictType.toString() = 'CONTRADICTING'"* is used to assign an appropriate style when both versions modify an element to contradictory values.

MiniJavaCompare supports styling further by automatically generating custom icons for meta-classes in *DSDiffCS* and *CRRuleCS* from an original icon in *CS* to support each condition. For each style of 21 conditional styles generated for `NodeMapping` and `ContainerNodeMapping`, it creates a unique icon. For example, as illustrated in Fig. 6.1, the icon used for the method *renewBook* has a red border to represent that it contains contradictory changes generated based on *conflictType*. It also contains tilde signs on both sides to show that both users have modified the values in the object, which is generated based on *diffKind*. Moreover, it uses blue color to show the operation of the left author and purple color to show the operation of the right author.

Similarly, for each `EdgeMapping`, we create nine conditional styles. These conditional edge styles define suitable colors and styles for the edges. The number of edge conditional styles is fewer than node conditional styles due to the fact that an edge cannot express a contradictory situation, and we use association diff classes for this purpose.

For the property page of the root meta-class of the *CS*, Package in *Mini-Java*, we automatically incorporate three commands and actions: *Save* for saving changes, *Close* for closing the model, and *Commit* for invoking *Algorithm 7* to address conflicts and perform merging.

However, for the property page of the root meta-class of the *DSDiffCS*, we automatically incorporate five commands: *Save* for saving changes, *Close* for closing the model, *Merge* for invoking the merge algorithm (*Algorithm 9*) to address conflicts and perform merging, *Hide details* to conceal fine-grained differences, and *Show details* to reveal fine-grained differences. Additionally, we introduce a *List* input to the properties page, which displays conflict resolution objects that are `Pending` and require attention from the DSL user.

Furthermore, on the property page of the conflict resolution meta-class within *DSDiffCS*, we automatically include a *Select* input. This input allows the DSL user to choose a resolution strategy, and a *Clear resolution* command is provided to undo any previously selected resolution made by the DSL user. When an option other than `NIL` is chosen from the resolution strategy *Select*, the associated conflict resolution object's color changes to *Green*, and the resolution type is set to *Resolved*. Selecting `NIL` or activating the *Clear resolution* command sets the resolution type back to *Pending* and changes the color to orange.

6.2. Layering the differences

One of the main concerns is to manage complexity when the number of elements in the *Diff*$_{012}$ model arises. To tackle this issue, we have introduced the layering mechanism provided by Sirius. In contrast to show/hide functions which deals with fine-grained differences, the layering function address the complexity and cognitive demands by hiding/showing the group of conflict resolutions, semantic differences, and semantic conflicts.

This allows DSL user to have focused visualization of elements of interest while reducing the verbosity associated with reported conflict resolutions. This division, also simplifies the maintenance and updating of specific layers for DSL users, all without exerting any influence on other layers, thereby enhancing the overall system's adaptability. Each layer possesses the capacity to concentrate on distinct facets or perspectives of the *DSDiffCS* model, thereby contributing to a more manageable and comprehensible representation.

7. Manual resolution

Some conflicts cannot be resolved without user intervention. However, effective manual conflict resolution requires certain features to be provided. DSMCompare offers a user interface mechanism and APIs designed to facilitate the manual resolution of conflicts that cannot be automatically resolved.

One of the main requirements is conflict visualization. Users need a clear and comprehensive representation of the conflicts present in the model. DSMCompare detects all conflicts and creates a conflict resolution object attached to each one. It provides visual cues by displaying pending conflict resolutions in orange and highlighting conflicting areas in red, helping users quickly identify areas of conflict.

For each conflict, users should be able to view the conflict description and access detailed information about the conflicting elements. DSMCompare automatically generates meaningful titles describing the conflicts. For example, DSMCompare connects each conflict resolution object to related semantic or fine-grained conflicts. In the case of semantic conflicts, users can also see the related semantic differences. This assists in better reasoning about the correct intention and, consequently, conflict resolution. In the case of fine-grained conflicts, users can see details such as the kind of differences and changed values from each side.

Users should be presented with various options for resolving conflicts. By selecting individual conflict resolution objects within the $Diff_{012}$ model, DSL users are provided with options for conflict resolution. These options include: *"Keep Left"*, *"Keep Right"*, *"Apply Left Then Right"*, *"Apply Right Then Left"*, *"Discard All changes"*, *"Keep Semantic Difference"*, and *"Custom"*. The *"Keep Left"* operation prioritizes differences from the *"LEFT"* version while ignoring differences from the *"Right"* version. Similarly, the *"Keep Right"* operation follows a similar pattern but prioritizes the *"Right"* version. Operations like *"Apply Left Then Right"* combine the actions of *"Keep Left"* and *"Keep Right"*, effectively resolving differences between both versions. The *"Apply Right Then Left"* operation mirrors this process. The *"Discard All changes"* operation, as the name implies, simply ignores all differences. On the other hand, the *"Keep Semantic Difference"* operation suggests resolving differences associated with the version where a semantic difference occurred, while ignoring changes related to the version with fine-grained differences.

The ability to undo is crucial in case a user changes their mind or realizes that the chosen resolution is not suitable. DSMCompare saves the user decisions in the $Diff_{012}$ model and does not apply them to the V_{012} model until all conflicts are marked as resolved. When the user changes the *status* of a conflict resolution object from `Resolved` to `Pending`, all changes are rolled back in the $Diff_{012}$ model. Additionally, Sirius inherits the built-in undo/redo functionality for the modeling workbench.

It is also important that users be able to apply custom changes to resolve conflicts. DSMCompare provides users with flexibility in resolving complex cases by allowing modifications to the $Diff_{012}$ model to accurately represent the intentions of both parties. This option allows changes to any desired element within the $Diff_{012}$ model, enabling the creation of specific changes. For existing unmerged Diff elements (where *isMerged* is set to false), users can apply modifications and set the *mergeKind* element to one of the values: *MERGE-ADD*, *MERGE-DELETE*, or *MERGE-MODIFY*, based on the custom change they have implemented.

Another significant feature is conflict marking, which DSMCompare offers through the use of *isMerged* and *isIgnore* attributes, along with relevant processes in Algorithm 9 and Algorithm 10. This ensures that once a conflict is resolved, it is marked as such to prevent unnecessary re-evaluation in subsequent manual resolution iterations.

Furthermore, DSMCompare provides conflict navigation. Users can navigate between conflicts efficiently through the conflict resolution group pane. Additionally, the list of remaining conflict resolution objects is available through a list on the properties page.

Moreover, DSMCompare offers guidance and documentation on conflict resolution strategies and different resolution options, helping users make informed decisions.

8. Model Merge

DSMCompare employs a systematic merge approach to resolve conflicting differences during domain-specific model integration. The process involves strategic decision-making to address conflicts that arise when combining model changes from various sources. A variety of *conflict resolution strategies* are utilized to ensure a seamless integration process. These strategies encompass retaining changes from one version while discarding conflicting ones, as well as applying modifications sequentially to maintain compatibility.

In addition to fine-grained conflicts, DSMCompare incorporates *semantic conflicts* into the evaluation process. The tool distinguishes between *automated* and *custom conflict resolution*. While automated mechanisms handle straightforward conflicts, complex scenarios often require human intervention. The inclusion of *custom resolution strategies* empowers DSL users to tailor conflict resolution to the unique needs of their projects.

8.1. Algorithm

Algorithm 9 The Resolve and Merge Algorithm

```
 1: procedure RESOLVE($Diff_{012}$)
 2:   $V_{012} \leftarrow LoadV012()$
 3:   for all $resolution$ in $Diff_{012}$ do
 4:     if $resolution.status = Resolved$ then
 5:       $diffList \leftarrow getConflictsNotMergedNotIgnored(resolution)$
 6:       switch $resolution.Strategy$ do
 7:         case "Keep Left"
 8:           $Merge(V_{012}, diffList, LEFT)$
 9:         case "Keep Right"
10:           $Merge(V_{012}, diffList, RIGHT)$
11:         case "Apply Left Then Right"
12:           $Merge(V_{012}, diffList, LEFT)$
13:           $Merge(V_{012}, diffList, RIGHT)$
14:         case "Apply Right Then Left"
15:           $Merge(V_{012}, diffList, RIGHT)$
16:           $Merge(V_{012}, diffList, LEFT)$
17:         case "Discard All changes"
18:           break
19:         case "Keep Semantic Difference"
20:           $semDiff \leftarrow findSemDiff(resolution)$
21:           $diffListInSemDiff \leftarrow getConflictsNotMergedNotIgnored(semDiff)$
22:           $Merge(V_{012}, diffListInSemDiff, semDiff.author)$
23:         case "Automatic" OR "Custom"
24:           $diffListMerge \leftarrow findDiffListMerge(resolution)$
25:           $Merge(V_{012}, diffList.Append(diffList + diffListMerge))$
26:       $SetMerged(diffList)$
27:     end if
28:   end for
29: end procedure
```

Algorithm 10 The General Merge Algorithm

```
Input: $V_{012}$ successive model version, diffList list of diffs, author the version
1: procedure MERGE($V_{012}$, diffList)
2:     sortedDifferences ← SORTDIFFERENCES($Diff_{012}$)
3:     for all diff in sortedDifferences do
4:         if not ISREALCONFLICTING(diff) OR (ISRESOLVED(diff)) then
5:             diff.applyTo($V_{012}$, author)
6:         end if
7:     end for
8: end procedure
```

Algorithm 9 outlines forms the cornerstone of the DSMCompare conflict resolution and merging process. It operates by harnessing the insights and decisions embedded within the *Diff*012 model to address conflicts and seamlessly incorporate changes into the evolving *V*012 model. This procedure embodies a comprehensive set of tasks, seamlessly transitioning through the steps guided by user choices.

The algorithm starts by initializing the model version *V*012 and traverses through each resolution stored within *Diff*012. The algorithm distinguishes between resolved resolutions and other instances. Upon encountering a resolved resolution, the algorithm endeavors to select appropriate strategies for conflict resolution.

The algorithm continues with identifying conflicts that remain unresolved, unmerged, and not ignored using the `getConflictsNotMergedNotIgnored()` function. Employing a `switch-case` structure, various strategies are selected. These strategies involve preserving either the left or right version, applying changes first in the left-then-right or in reverse order, discarding changes, addressing fine-semantic conflicts by preserving semantic differences, or handling automatic and custom resolutions.

To execute these strategies, specific functions are employed. `findSemDiff()` identifies semantic differences for a resolution, especially when the strategy focuses on maintaining semantic differences. `getConflictsNotMergedNotIgnored()` collects unresolved conflicts. The `Merge()` function calls *Algorithm 10* to introduces changes to the V_{012} model, guided by the chosen strategies.

Subsequently, the algorithm is applied to all resolutions in $Diff_{012}$, using strategies as required. At the end of each iteration, the `SetMerged()` functions is called to assign a *true* to the *isMerged* attribute of the difference object. This prevents resolved conflicts from being reconsidered in subsequent iterations or function calls.

The algorithm described in *Algorithm 10* outlines the process of merging changes within a model, incorporating a series of modifications from a given list of differences, *diffList*, into a final merged model, V_{012}. The algorithm iterates through each difference, denoted as *diff*, within the provided list.

For each *diff* in *diffList*, the algorithm checks if the *diff* should be ignored, as determined by the condition *diff.isIgnore = False.* If the condition is met, the algorithm proceeds to determine whether the *diff* is in conflict or has already been resolved.

If the *diff* is either not in conflict (*diff.conflict = null*) or is in conflict but has already been resolved (*diff.conflict = "RESOLVED"*), the algorithm proceeds to apply the changes introduced by the *diff* to the *V*012 model. Using the *applyTo* method, it modifies the *V*012 model based on the information stored within the *diff* object associated with the respective author. If the author is *RIGHT* the changes are applied to the right version, otherwise, it applies to the left version. This process ultimately results in a final merged model, ensuring that conflicts are appropriately managed and resolved changes are applied to the merged result.

9. Evaluation

We assess the performance of our approach using model histories created by third parties. The breakdown of the evaluation setup for domain-specific conflict resolution and model merge is as follows: Section 9.1 delves into the technical details of how the project was executed, including the tools and methodologies used. Section 9.2 defines the intended goals and objectives. Section 9.3 outlines the methodology for any experiments conducted during the evaluation. Section 9.4 presents the findings and outcomes of these experiments. Section 9.6 discusses potential limitations and challenges in the evaluation process, and Section 9.5 provides a critical analysis and interpretation of the results and their implications.

9.1. Implementation

We implemented DSMCompare as an Eclipse plug-in designed to run on the Eclipse Modeling Framework (EMF version 2022-06) and Sirius (version 7). The tool is available for download through the open-source repository.[3] To identify generic model-based matches and

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

differences, we instantiate the CDOCompare engine,[4] which is one of the default EMFCompare engines in Eclipse. We rely on the API of EMFCompare to retrieve the difference set between the three versions. Using Java, DSMCompare transforms these generic differences into an instance of the `DSDiffMM` metamodel using the EMF API. To create an executable form of SDRules and CRRules, we use Xtend to transform domain-specific rules into Henshin textual format.[5] Next, we execute the Henshin version of the SDRules to enhance the *Diff*$_{012}$ model with semantic differences. Following that, we calculate the semantic conflicts. Finally, we resolve conflicts according to Algorithm 7 as presented in Section 4.

9.1.1. *Merge editor*

Fig. 6.9 displays the *Diff*$_{012}$ model editor. This editor illustrates fine-grained conflicts, semantic conflicts, fine-grained differences, semantic differences, and conflict resolutions. The editor also contains a palette including all the elements that can be added to the *Diff*$_{012}$ model. Using the palette, DSMCompare accommodates manual (custom) resolution, granting users the autonomy to tailor conflict resolutions according to specific requirements.

[4]`https://www.eclipse.org/cdo`, last accessed August 2023
[5]`https://www.eclipse.org/xtend/index.html`, last accessed August 2023

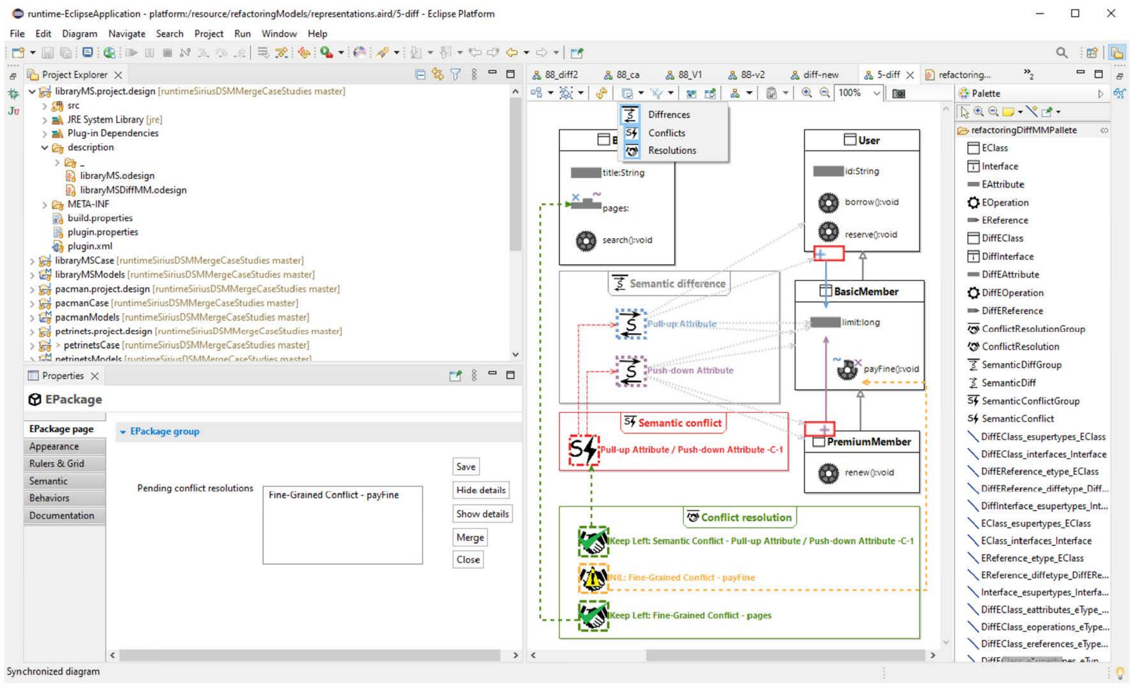

runtime-EclipseApplication - platform:/resource/refactoringModels/representations.aird/5-diff - Eclipse Platform
File Edit Diagram Navigate Search Project Run Window Help
Project Explorer
libraryMS.project.design [runtimeSiriusDSMMergeCaseStudies master]
src
JRE System Library [jre]
Plug-in Dependencies
description
libraryMS.odesign
libraryMSDiffMM.odesign
META-INF
build.properties
plugin.properties
plugin.xml
libraryMSCase [runtimeSiriusDSMMergeCaseStudies master]
libraryMSModels [runtimeSiriusDSMMergeCaseStudies master]
pacman.project.design [runtimeSiriusDSMMergeCaseStudies master]
pacmanCase [runtimeSiriusDSMMergeCaseStudies master]
pacmanModels [runtimeSiriusDSMMergeCaseStudies master]
petrinets.project.design [runtimeSiriusDSMMergeCaseStudies master]
petrinetsCase [runtimeSiriusDSMMergeCaseStudies master]
Properties
EPackage
EPackage page
Appearance
Rulers & Grid
Semantic
Behaviors
Documentation
EPackage group
Pending conflict resolutions
Fine-Grained Conflict - payFine
Save
Hide details
Show details
Merge
Close
88_diff2
88_ca
88_V1
88-v2
diff-new
5-diff
refactoring...
Diffrences
Conflicts
Resolutions
title:String
pages:
search():void
User
id:String
borrow():void
reserve():void
Semantic difference
Pull-up Attribute
Push-down Attribute
BasicMember
limit:long
payFine():void
Semantic conflict
Pull-up Attribute / Push-down Attribute -C-1
PremiumMember
renew():void
Conflict resolution
Keep Left: Semantic Conflict - Pull-up Attribute / Push-down Attribute -C-1
NIL: Fine-Grained Conflict - payFine
Keep Left: Fine-Grained Conflict - pages
Palette
refactoringDiffMMPallete
EClass
Interface
EAttribute
EOperation
EReference
DiffEClass
DiffInterface
DiffEAttribute
DiffEOperation
DiffEReference
ConflictResolutionGroup
ConflictResolution
SemanticDiffGroup
SemanticDiff
SemanticConflictGroup
SemanticConflict
DiffEClass_esupertypes_EClass
DiffEClass_interfaces_Interface
DiffEReference_etype_EClass
DiffEReference_diffetype_Diff...
DiffInterface_esupertypes_Int...
EClass_esupertypes_EClass
EClass_interfaces_Interface
EReference_etype_EClass
EReference_diffetype_DiffERe...
Interface_esupertypes_Interfa...
DiffEClass_eattributes_eType_...
DiffEClass_eoperations_eType...
DiffEClass_ereferences_eType...
Synchronized diagram

Figure 6.9. The *Diff*$_{012}$ editor

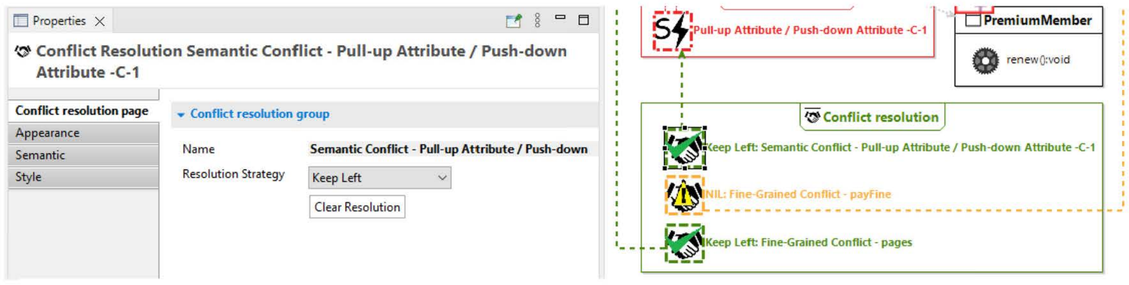
The properties view allows the DSL expert to navigate between conflict resolutions through the list box that shows remaining conflict resolutions. By clicking on the *Save* and *Close* button, users can save partial resolutions. This functionality enables users to pause the merging process and later resume from where they left off, minimizing disruptions and facilitating effective conflict management. The *Merge* button starts the conflict resolution and merges the procedure if the list of remaining conflict resolutions is empty; otherwise, it shows a message asking the user to resolve all the conflicts before proceeding to merge. Users can also use the *Hide details* option to filter out fine-grained details or *Show details* to investigate the details of semantic changes or semantic conflicts.

Additionally, the *Layers* mechanism shown on top of the figure allows hiding and showing entire layers of semantic differences, conflicts, or conflict resolutions. Each layer has the capacity to focus on distinct facets or perspectives of the $Diff_{012}$ model, contributing to a more manageable and comprehensible representation.

Fig. 6.9 displays the resolution reversibility mechanism that appears when the DSL user selects a conflict resolution object in the editor. The *Clear Resolution* button undoes previously made resolution decisions (sets the resolution strategy to *NIL* value) and all the fine-grained changes associated with the resolution. The editor also allows iterating on the available resolution strategies, refining the resolution as the merging process unfolds.

9.1.2. *Usage scenario*

Whenever conflicts require user intervention to resolve them during the process of pushing changes to merge, DSMCompare automatically triggers and automatically initiates and displays the content of the $Diff_{012}$ model, which contains automatically generated pending conflict resolution objects organized in a dedicated list. Users can seamlessly select pending


Properties
Conflict Resolution Semantic Conflict - Pull-up Attribute / Push-down Attribute -C-1
Conflict resolution page
Appearance
Semantic
Style
Conflict resolution group
Clear Resolution
Pull-up Attribute / Push-down Attribute -C-1
PremiumMember
renew():void
Conflict resolution
Keep Left: Semantic Conflict - Pull-up Attribute / Push-down Attribute -C-1
NIL: Fine-Grained Conflict - payFine
Keep Left: Fine-Grained Conflict - pages

Name: Semantic Conflict - Pull-up Attribute / Push-down
Resolution Strategy: Keep Left

Figure 6.10. Undoing a resolution

conflict resolution objects either directly from the model or from the list. These pending conflict resolution objects are initially highlighted with an orange theme, making them easily identifiable. When the user selects a conflict resolution strategy, it becomes completed and changes to a green theme.

If the user clears a previously selected resolution, DSMCompare undoes all the relevant changes required to resolve the conflict. Consequently, the conflict resolution theme turns back to orange, and the conflict resolution is added back to the pending list. DSMCompare does not proceed with the merge until all conflicts are resolved, and the list of pending conflict resolutions is empty. Any attempt to merge prompts the user with an appropriate notification.

Once all the conflicts are resolved, the user initiates the merge process. DSMCompare applies all the changes related to conflict resolutions in the *Diff*012 to the V012 model. Subsequently, it creates a new commit incorporating the resolved changes and pushes the new commits back to the shared repository.

9.2. Objectives

As we have already evaluated, DSMCompare effectively detects differences and conflicts (Zadahmad et al, 2019, 2022). Here, we evaluate the effectiveness of DSMCompare when resolving merge conflicts. Therefore, in this experiment, we assess DSMCompare with respect to the following research questions:

RQ1 *Does DSMCompare resolve conflicts correctly?*
Using a public dataset as benchmark, we verify if DSMCompare resolves conflicts similarly to the baseline, misses resolutions and incorrectly resolves conflicts.

RQ2 *To what extent does DSMCompare reduce the amount of manual user interactions?*
DSMCompare is uses computes differences and merges between models at the model-level rather than lines of code and uses semantic rules. Our hypothesis is that users using such an approach will face fewer conflicts to resolve and merge models with fewer steps than approaches not using semantic (like EMFCompare) and not using model-based (like Git) differencing/merging.

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*

In the absence of existing repositories containing domain-specific models and their associated semantic difference rules, our experiment focuses on Ecore models reverse-engineered from Java programs. As described in (Zadahmad et al, 2022), we define a refactoring pattern as a semantic difference. If the refactoring pattern plays a role in a conflict, it is considered as a semantic conflict. The conflict resolution objects generated automatically for both fine-grained and semantic conflicts. Whenever a conflict needs to be resolved, we create a conflict resolution object for both fine-grained and semantic conflicts.

We use open-source code-based projects developed in Java as our reference and convert them into Ecore models. This allows us to analyze their histories for refactoring modifications and run our domain-specific conflict resolution and merge experiments.

We have selected a total of 95 Java-based software repositories from GitHub using Random Forest classifiers, as described in (Munaiah et al, 2017). The selection criteria for these repositories are: they needed to have (i) communities consisting of two or more contributors and (ii) a minimum of 500 stars on GitHub. To narrow our focus to repositories with merge conflicts, we employ the *RefConfMiner* project (Shen et al, 2019). This helps us create a database that includes commit IDs related to refactoring-induced merge conflicts, along with identifiers for version triplets (V0, V1, V2) and the detected refactoring types.

Out of the initial 95 repositories, only 13 projects met the criteria, containing at least three refactoring-related merge conflicts that could be processed with *RefactoringMiner* and downloaded successfully. These final selected projects are: `android`, `closure-compiler`, `error-prone`, `jabref`, `junit4`, `mcMMO`, `POSA-14`, `querydsl`, `realm-java`, `redpen`, `storm`, `syncany`, and `titan`.

Subsequently, we use *MergeScenarioMiner* (Shen et al, 2021) to collect the Java files involved in conflicting merge commits. This process generates a separate folder for each conflicting commit ID, with five sub-folders: one containing the parts of the project related to V0, another containing the changed parts in V1, a third one for the changes in V2, a fourth one for the result of the Git merge, and a fifth one for the manual merge done by the user. We refer to the latter two results *gitMerged* and *manualMerged*, respectively.

Next, we convert the source codes in each folder into instances of the Knowledge Discovery Metamodel (KDM) using Eclipse's *MoDisco* framework (Bruneliere et al, 2014). These KDM instances are subsequently transformed into instances of MiniJava models. Note that conflict markers are ignored when creating the model for *gitMerged*.

In addition, we manually prepare 17 semantic difference rules to encapsulate the refactorings. These rules are adapted from the experiment in (Zadahmad et al, 2022) and tailored to fit the MiniJava metamodel. An example of such an SDRule designed for this experiment is depicted in Fig. 6.4. We also manually prepare nine CRRules to automatically or semi-automatically resolve conflicts. These rules are adapted from different recent researches on the domain of refactoring (Ellis et al, 2023) and tailored to fit the MiniJava metamodel. An example of such an SDRule designed for this experiment is depicted in Fig. 6.7a.

9.3.2. *Methodology*

To answer *RQ1*, we compare the conflict resolutions found manually in the dataset, with the conflict resolution performed by DSMCompare. For each commit i, we calculate the difference between *gitMerged* and *manualMerged* using DSMCompare. This difference encompasses a collection of fine-grained and semantic changes made by the user to resolve the conflicts manually. We refer to this difference as the set crG_i. It is the baseline to address *RQ1*.

We follow a specific workflow for each conflicting commit as outlined in Algorithm 7 in Section 4. We begin by obtaining three instances of the MiniJava metamodel (V0, V1, V2) for each conflicting commit and subsequently calculate differences and conflicts with DSMCompare. In cases where all changes are trivial or can be resolved by predefined conflict resolution rules, DSMCompare generates the successive model (*Diff*012 model). However, if there are remaining conflicts that require user intervention, DSMCompare initiates the *Diff*012 model, which contains automatically generated pending conflict resolution objects organized in a dedicated list. We open the $Diff_{012}$ model whenever we feel comfortable resolving conflicts.

We can select pending conflict resolution objects directly from the model. These pending conflict resolution objects are initially highlighted with an orange theme, making them easily identifiable. If we choose a conflict resolution strategy and clear it, DSMCompare undoes all the relevant changes required to resolve the conflict. As a result, the conflict resolution theme reverts to orange from green, and the conflict resolution is added back to the pending list for further user attention.

DSMCompare does not proceed with the merge until all conflicts are resolved, and the list of pending conflict resolutions becomes empty. Any attempt to merge prompts us with an appropriate notification, explaining the need to address the remaining conflicts beforehand.

Once all the conflicts are resolved, we initiate the merge process. DSMCompare applies all the changes in the *Diff*012 model related to conflict resolutions to create the successive merged version, the *V*012 model.

We refer to the set of conflict resolutions applied by DSMCompare as *crD*. Finally, we compare the results of *crG* with those of *crD* to evaluate the effectiveness of DSMCompare in conflict resolution.

We denote the two sets crG_i and crD_i for each commit i, respectively. We rely on precision and recall measures for the comparison. We define four variables as follows for each commit i:

- **Correct resolutions** $CR_i = crD_i \cap crG_i$ is when DSMCompare resolves conflicts in the same way as in Git.
- **Incorrect resolutions** $IR_i = crD_i \setminus crG_i$ is when DSMCompare incorrectly resolves conflicts with respect to Git. The correct conflict is resolved, but not in the same way.
- **Missed resolutions** $MR_i = crG_i \setminus crD_i$ but only for those that DSMCompare ommits to resolve a conflict that it should have as indicated in Git.
- **Correct misses** $CM_i = crD_i \setminus crG_i$ but only for those that DSMCompare correctly ommits to resolve conflcits because they do not occur in a model-based differencing/merging scenarios.

It follows that precision and recall are defined for each commit i as:

$$precision_i = \frac{CR_i}{CR_i + IR_i} \tag{9.1}$$

$$recall_i = \frac{CR_i}{CR_i + MR_i} \tag{9.2}$$

In Equation (9.1), we define precision as the ratio between the correctly found semantic resolutions in DSMCompare for each conflicting commit and the total number of semantic conflict resolutions it finds across those conflicting commits. In Equation (9.2), we define recall as the ratio between the correctly found semantic resolutions in DSMCompare for each conflicting commits and the expected number of semantic conflict resolutions found manually across those conflicting commits.

To address *RQ2*, we estimate the amount of manual work required to resolve conflicts using DSMCompare, EMFCompare, and Git on the same dataset.

We denote mD_i as the amount of resolutions (merge) performed manually when using DSMCompare for each commit i. We consider all resolution strategies presented in Section 3: when users (i) select the appropriate resolution strategy and when they (ii) resort to a custom resolution. In mD_i, we consider both resolutions of semantic and fine-grained conflicts. For (i), we count each selection individually. For (ii), we count the number of changes the user performs: the number of elements with change status *MERGE-ADD*, *MERGE-DELETE*, and *MERGE-MODIFY*.

Similarly, we denote mE_i as the amount of resolutions performed manually when using EMFCompare for each commit i. EMFCompare offers two main strategies to resolve conflicts: *Keep Left* and *Keep Right*. We include in mE_i the selection of a strategy for (i) fine-grained conflicts and (ii) any custom or manual edits made outside of EMFCompare on the subsequent version, since EMFCompare does not permit editing during the conflict resolution process. For (i), we count each selection individually. For (ii), we count the number of changes the user performs.

Finally, we denote mG_i as the amount of resolutions performed manually for each commit i when using a textual diff/merge tool (Git). It is unclear which tool users employed to resolve the conflicts manually in the dataset we collected. Therefore, we assume that they performed only one manual action to resolve each textual conflict, representing the minimum number of actions. This assumption implies that we are providing a significant discount to textual merging: in practice, users may need to perform multiple custom edits to resolve each conflict.

To answer RQ2, we compute the reduction of manual merges needed as follows:

$$merge\_reduction_{DE} = \frac{mE_i - mD_i}{mE_i} \tag{9.3}$$

$$merge\_reduction_{DG} = \frac{mG_i - mD_i}{mG_i} \tag{9.4}$$

Equation (9.3), calculates the reduction in work when using DSMCompare compared to EMFCompare for each commit. Equation (9.4), calculates the reduction in work when using DSMCompare compared to manual conflict resolution during the text-based merging for each commit.

It is noteworthy to express that, we manually compare each resolution output in DSMCompare with those in EMFCompare and with manually resolved conflicts. For each refactoring or conflict reported by DSMCompare, we analyze the report, which includes the Java file, the *gitMerged* model, the *manualMerged* model, the EMFCompare results, and the names of the involved elements (e.g., package, class, method, attribute). Subsequently, we manually compare this information with the results in the corresponding Diff_{012} model generated by DSMCompare.

9.4. Experiment results

9.4.1. *Characterization of the resulting dataset*

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

Table 6.1. Characteristics of dataset

Number of	
Projects	13
MiniJava models in Ecore	444
Semantic difference (refactoring) rules	14
Commits	96
Commits without model-based semantic conflicts	22
Commits without manualMerged folder	6
Total number of	
Semantic Difference	2 472
Semantic Conflicts	533
Fine-grained diffs	12 006
Conflicting fine-grained diffs	1 531
DSMCompare	
Semantic conflict resolutions - Automatic	426
Semantic conflict resolutions - Manual	34
Fine-grained conflicting Diffs - Automatically resolved	657
Fine-grained conflicting Diffs - Manually resolved	60
- by applying Semantic conflict resolutions - Manual	
Fine-grained conflicting Diffs - Manually resolved	814
Semantic conflict resolutions - Manual strategy choice	34
Semantic conflict resolutions - Manual custom	12
Fine-grained conflict resolutions - Manual strategy choice/custom	814
EMFCompare	
Fine-grained conflict resolutions - Manual strategy choice	1 857
Fine-grained conflict resolutions - Manual custom (edit)	46
Text-based	
Fine-grained conflict resolutions - Manual strategy choice/custom	2 091

Table 6.2. Distributions per commit across all projects

Text	Median	Mean	Std. Dev.
	DSMCompare		
Conflicting fine-grained diffs	6	20.41	33.59
Semantic conflict resolutions - Automatic	2	6.26	9.62
Semantic conflict resolutions - Manual	0	0.5	1
Fine-grained conflicting Diffs - Automatically resolved	3	8.76	14.04
Semantic conflict resolutions - Manual strategy choice	0	0.53	1.02
Semantic conflict resolutions - Manual custom (edit)	0	0.19	0.39
Fine-grained conflict resolutions - Manual strategy choice/custom	3.50	12.72	23.57
	EMFCompare		
Fine-grained conflict resolutions - Manual strategy choice	6	24.76	39.53
Fine-grained conflict resolutions - Manual custom (edit)	0	0.53	1.24
	Text-based		
Fine-grained conflict resolutions - Manual strategy choice/custom	8	27.88	44.79

In total, we generate 96 sets of six Ecore models: the three model versions of every commit (V0, V1, V2), the *gitMerged* model, the *manualMerged* model, and the Diff_{012} model including the differences, conflicts, and conflict resolutions. Out of the 87 refactoring types (Tsantalis et al, 2020), we found only 14 semantic difference rules repeated in the Diff_{012}

models. The complete dataset is available online [6]. Table 6.1 provides a comprehensive overview of the characteristics of the dataset used in the evaluation, as well as the conflict resolution results obtained from DSMCompare, EMFCompare, and the text-based manual approach.

The results demonstrate that DSMCompare can automatically resolve a large number of semantic and fine-grained conflicts. Note that, since we downloaded only the Java files involved in the conflicting commits for all three versions, the $Diff_{012}$ model represents only the minimal model needed to comprehend the context of the changes in each version, as opposed to displaying the complete models of the entire Java code.

We excluded six commits and their associated models from our calculations because we were unable to download the related *manualMerged* folder. Additionally, we omitted 22 commits and their associated models from our semantic-related calculations due to inherent differences in how text-based and model-based VCS systems operate and how they interpret and detect conflicts. DSMCompare correctly identifies cases that are incorrectly reported as conflicts, often attributed to flaws in text-based merging tools, such as changes to the organization of code (*e.g.,* moving methods or classes). We also excluded commits in which all of their refactoring types contributing to the conflicts were related to changes in method bodies that are not explicitly modeled in MiniJava (*e.g., Inline method*).

The total number of semantic differences identified in the dataset is 2 472 and the total number of semantic conflicts is 533. Additionally, fine-grained differences amount to 12 006, with 1 531 of them being conflicting. These numbers demonstrate the complexity and diversity of changes and conflicts present in the dataset.

DSMCompare automatically resolved 426 semantic conflicts (81% of semantic conflicts) and 657 fine-grained conflicting differences (55% of conflicting fine-grained differences). Moreover, DSMCompare's manual conflict resolution feature was used to resolve 34 semantic conflicts (6% of semantic conflicts) which leads to resolving 60 fine-grained conflicts (4% of conflicting fine-grained differences) manually. Moreover, an additional 814 fine-grained conflicts were resolved manually, emphasizing the tool's efficacy in handling both automatic and manual conflict resolution scenarios. In contrast, EMFCompare successfully handled

[6] `https://zenodo.org/deposit/8333378`

the equivalent of 1 857 generic model-based fine-grained conflicts using manual choice of resolution strategy and 46 using custom edits in the successive version model. Text-based tools were also employed, resulting in the manual resolution of 2 091 equivalent textual fine-grained conflicts using a combination of manual strategy choice and custom edits.

Table 6.2 provides a detailed breakdown of conflict resolution statistics across different tools, with a focus on median, mean, and standard deviation values. When comparing the performance of DSMCompare, EMFCompare, and Text-based tools, it is evident that DSMCompare generally requires fewer manual interventions for conflict resolution. For instance, DSMCompare shows a median of 0 for both *Semantic conflict resolutions - Manual* and *Semantic conflict resolutions - Manual strategy choice*, indicating a significant reduction in manual resolution efforts compared to EMFCompare and Text-based tools.

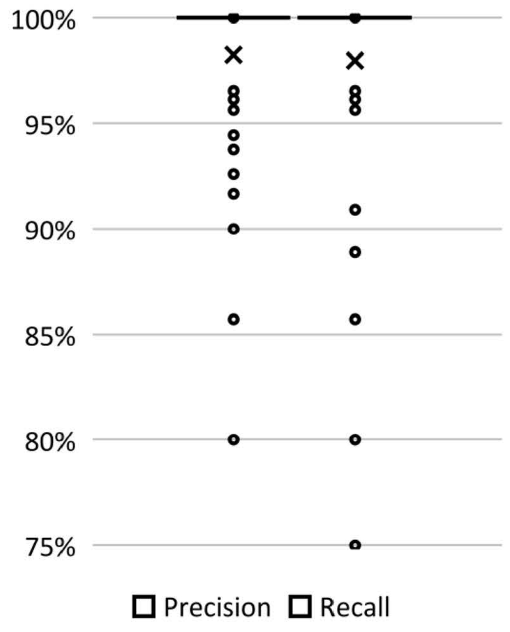
In contrast, EMFCompare exhibits slightly higher median values for fine-grained conflict resolutions, suggesting a relatively higher manual intervention requirement compared to DSMCompare. Text-based tools, on the other hand, have the highest median values among the three tools, indicating a greater reliance on manual conflict resolution strategies. Furthermore, DSMCompare boasts a lower standard deviation across various conflict resolution categories, implying a more consistent and predictable performance in handling conflicts. EMFCompare and Text-based tools exhibit higher standard deviations, suggesting greater variability in their conflict resolution outcomes.

Overall, these statistics highlight that DSMCompare tends to outperform EMFCompare and Text-based tools in terms of efficiency and consistency in conflict resolution, with significantly fewer manual interventions required.

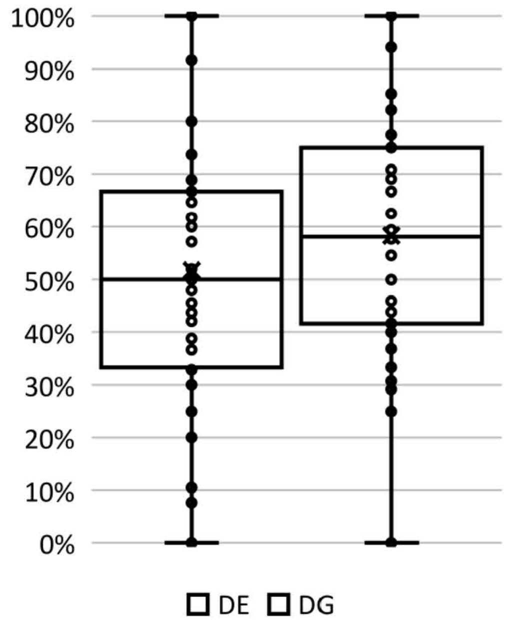
9.4.2. *Correctness of DSMCompare semantic conflict resolutions*

Fig. 6.11a presents the distribution of the precision and recall of conflict resolutions for semantic conflicts identified by DSMCompare across the 68 commits of the dataset. These results are computed in accordance with equations (9.1) and (9.2), as the reference baseline for comparison. These equations specifically focus on semantic conflict resolutions, encompassing conflicts that involve semantic and semantic-fine differences, including refactorings.

The overall trends reflected in the box plots exhibit almost perfect scores, consistently hovering at a minimum of 98%. Notably, there is minimal variability in both precision and



(a) Precision and recall of conflict resolutions for semantic conflicts



(b) $merge\_reduction_{DE}$ and $merge\_reduction_{DG}$

Figure 6.11. Box-plots showing the distributions of the metrics across all commits

recall of conflict resolutions. The standard deviations stand at 4% and 5%, respectively, while the interquartile range is 0. These near-perfect scores demonstrate DSMCompare's ability to accurately resolve nearly all the semantic conflicts reported by Git.

Occasionally, false negatives (MR_i) do arise when DSMCompare misses one or two differences within some $Diff_{012}$ models containing a sparse number of conflict resolutions. For instance, in the case of commit `84094c7` of the `realm-java` project, *Manual conflict resolution* identifies four conflict resolutions, whereas DSMCompare detects only three, resulting in a precision and recall of 75%. Most of these situations pertain to text-based refactorings, where the refactoring is initially recognized inaccurately, leading to an erroneous resolution.

In rare instances where DSMCompare erroneously identifies a conflict resolution (IR_i), these situations typically involve refactorings associated with the body of a method, which our dataset records as unstructured strings. As an example, the *Inline method* refactoring type entails transferring the content of one method to another calling it.

9.4.3. *Reducing user intervention during merge*

Fig. 6.11b reveals substantial reductions in the amount of user intervention required to resolve conflicts when using DSMCompare. When compared to EMFCompare, DSMCompare acheives at least 50% reduction in half of the commits. Notably, in projects such as *mamute*, *error-prone*, and *redpen*, DSMCompare achieved 96% reduction in user involvement, underlining its proficiency in automating conflict resolution tasks. These findings suggest that DSMCompare, built upon a model-based version control system, is particularly adept at handling conflicts in a manner that minimizes manual effort, offering a significant advantage over generic model-based approaches.

Moreover, when comparing with manual text-based conflict resolution as illustrated in Fig. 6.11b, DSMCompare reduces even better results with at least 58% merge reduction. This indicates that DSMCompare presents a compelling alternative to the labor-intensive manual conflict resolution process.

9.5. Discussion

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

9.5.1. *RQ1: DSMCompare effectiveness to resolve semantic conflicts*

According to the results, DSMCompare can find almost all conflict resolutions across all commits. It produces conflict resolutions for fine-grained and semantic conflicts of different types across different conflicting commits and model sizes in various projects.

However, DSMCompare was unable to find a few textual refactoring patterns that require investigating structural content encoded as strings that misled the conflict resolution. For the same reason, it also incorrectly detected a few refactorings which misled the conflict resolution.

9.5.2. *RQ2: DSMCompare effectiveness to reduce manual intervention*

According to the results, DSMCompare can reduce manual intervention compared to generic model-based and text-based VCS substantially. The data analysis reveals the remarkable efficiency of DSMCompare in simplifying conflict resolution processes across diverse software projects.

These findings underscore DSMCompare's capability to effectively automate conflict resolution tasks. This suggests that DSMCompare not only outperforms EMFCompare but also provides a valuable alternative to labor-intensive manual conflict resolution processes, potentially enhancing collaboration efficiency and reducing users' workloads in various software development projects.

9.5.3. *Advantage of using domain-specific model merging*

DSMCompare produced some true negatives (CM_i) because of structural changes which led to a decrease in conflict resolution efforts. For example, if one user moves a block of code to a different location within a file, and another user modifies that same block in the same file, a conflict is likely to occur in text-based VCS. However, in a DSMCompare, structural changes are often represented as higher-level operations (*e.g.,* pull-up, push-down, and moving), and DSMCompare understands that these changes are compatible, as they do not directly overlap in terms of model elements. Another example is when, in text-based VCS, if two users work on the same method and simultaneously extract a portion of the code into a new method, a conflict occurs in the shared method. However, in DSMCompare, when users extract methods in separate branches, DSMCompare recognizes the extraction operations and applies them without conflicts.

DSMCompare also prevents some conflicts regarding high-level abstractions using transitive conflict resolution rules. For example, if one user performs a pull-up method refactoring to move a method from a subclass to a superclass, and another user modifies the same method in the subclass, a conflict arises in text-based VCS due to the method relocation. However, DSMCompare recognizes the transitive change and handles it without flagging it as a conflict. As a result of the true negatives, in this evaluation, 22 conflicting commits are completely removed from conflict resolution activities and in the remaining conflicting commits, we have a decrease in the amount of effort needed for conflict resolution.

Furthermore, DSMCompare offers a more tailored display of conflict resolutions, and it is less verbose compared to Git and EMFCompare. Additionally, it is important to mention that DSMCompare does not necessitate users to create an ad-hoc metamodel. They can readily provide the metamodel of their DSL to utilize the tool. It explicitly associates the semantic conflict resolution instances to relevant conflicts and differences providing a comprehensive view of conflict and its resolution. It reports the conflict resolution using the original DSL concrete syntax. Therefore, DSMCompare helps to understand and locate the resolution that need to be done to 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.

9.6. Threats to validity

We outline some limitations of the experiment and our approach.

9.6.1. *Threats to internal validity*

Threats to the internal validity of this experiment are related to the assumptions we rely on. The reliance on multiple tools, including EMFCompare, for various stages of the conflict resolution process, introduces complexity. If the choice of projects or commits was influenced by the availability or compatibility of these tools, it could create selection bias. For instance, if certain projects were chosen because they work well with the toolchain, they might not be representative of cases where the toolchain is less effective.

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 with DSMCompare correspond to those found in *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.

To mitigate selection bias, we also carefully selected a diverse range of software projects from different domains and commits for evaluation, ensuring that our choices were not influenced solely by tool compatibility.

9.6.2. *Threats to construct validity*

Depending on DSL users to define SDRules for detecting semantic differences and conflict resolution introduces potential construct validity challenges. Variability in how users define these rules can affect the operationalization of conflict resolution outcomes. Inconsistent rule definitions may lead to ambiguous or subjective measurements.

The editors automatically generated by DSMCompare for the DSL engineers to specify SDRules completely reuse the abstract and concrete syntaxes of the DSL. Therefore, the editors are very familiar with the DSL expert to define the rules that mitigate the threat. One could also provide comprehensive guidelines and templates to DSL users for SDRule and CRRule definition.

Moreover, relying on user-defined rules and inputs, such as concrete syntax, can introduce measurement bias. If users' inputs are inconsistent or incomplete, it may impact the accuracy of conflict resolution metrics. Additionally, if there's a lack of clarity in how rules are defined, it can introduce measurement bias. Such problems particularly are likely, when the user deals with rules based on textual attributes such as method body like *Inline Method* which are hard to capture in the rules. This prevents DSMCompare from correctly detecting these refactoring (false negatives). Nevertheless, as the overall results show, DSMCompare can propose effective resolutions for every other refactoring type and conflict related to classes, associations, methods, and attribute changes.

9.6.3. *Threats to external validity*

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 and CRRules, 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 and their resolution. Our solution was to consider source code as models by reverse engineering repositories with these specificities.

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. Graphical visualization of differences hits its limits when the models have a lot of elements.

The conflict resolution that DSMCompare finds strongly depends on the CRRules provided by the DSL engineer. Thus, DSMCompare is only effective in providing conflict resolutions 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 CRRules from two successive versions exhibiting a semantic change, which further helps DSL engineers.

We do not claim that the results of the experiment optimized to identify refactoring opportunities in programs. Nevertheless, in the given dataset, DSMCompare can provide refactoring conflict resolutions on Ecore models that represent Java code.

10. Related Work

Sharbaf et al (2022b) conducted a systematic mapping study on conflict management techniques, versioning, and merging models. It discusses open issues and directions for future work which we found overlapping with our current work that we present in the following subsections.

10.1. Semantic Conflict Management

The conflict pattern language introduced by Sharbaf et al (2020) is used to express conflicts in different modeling languages. However, it relies on OCL and is limited to UML-based languages, which might be difficult for DSL engineers not familiar with them. Some methods (e.g., (Sharbaf et al, 2015)) handle specific instances of semantic conflicts, but lack flexibility for diverse conflicts across modeling languages, making managing semantic conflicts a challenge in model merging.

An approach in (Altmanninger et al, 2010) focuses on modeling language semantics and can detect semantic conflict. However, it relies on defining semantic views and representing models to introduce specific semantic aspects (Altmanninger, 2007). Other methods (e.g., (Sharbaf and Zamani, 2020; Dam et al, 2016)) mainly address static semantic conflicts and are tailored to specific modeling languages, limiting their use.

In DSMCompare, we extract complex change patterns from low-level model changes, similar to semantic lifting techniques like those in (García et al, 2013; Vermolen et al, 2012). However, their patterns are predefined and general, unlike our method's rules. DSMCompare generates an editor for the DSL, allowing DSL engineers to define semantic conflicts.This doesn't require expertise in complex languages like OCL, offering a domain-specific way to define constraints.

10.2. Visualization for Conflict Management Activities

An important drawback of current approaches lies in their lack of adequate visualization techniques for various conflict management tasks. Visualizing conflicts in the concrete syntax of different models is a significant challenge that hasn't been addressed for any modeling language (Sharbaf et al, 2022b).

Only a couple of existing methods (e.g., (Wieland et al, 2013; Bartelt and Schindler, 2010)) offer graphical support for conflict resolution, but they fall short of providing a clear overview of the model elements involved in conflict situations (Sharbaf et al, 2022b). Additionally, the visualization approach introduced in (Wieland et al, 2013) only supports manual resolution and focuses solely on fine-grained conflicts. Moreover, the approach in (Bartelt and Schindler, 2010) lacks support for various conflict resolution strategies, and it doesn't provide detailed information on specifying and Visualizing the resolution. This indicates a trend towards graphical support in conflict management (Sharbaf et al, 2022b). Hence, a promising avenue for future research could involve visually guiding users in describing conflict specifications.

Furthermore, adding a graphical representation of changes in the concrete syntax editor could enhance collaborators' awareness, helping them avoid conflicts during the modeling phase. However, there are only a few approaches that concentrate on conflict visualization (e.g., (Brosch et al, 2012f)) or provide user-friendly graphical editors (e.g., (Mens et al, 2005; Barrett et al, 2011)) for managing conflicts during model merging. Nonetheless, (Brosch et al, 2012f) lacks an editor to specify and visualize conflict and semantic resolution patterns. Moreover, the visualization method in (Mens et al, 2005) relies on cross tables, and the approach in (Barrett et al, 2011) utilizes text-based conflict reports. Thus, the ongoing evolution of graphical and visual solutions for conflict management remains a central challenge in this field (Sharbaf et al, 2022b).

DSMCompare offers a solution by generating a DSL editor. This enables DSL engineers to specify both semantic conflicts and conflict resolution patterns, all visualized using the DSL's original concrete syntax. DSMCompare visualizes both automatic and manual conflict resolution types. Clear styling distinguishes between automatic and custom changes, aiding in understanding the logic behind changes. DSMCompare also allows users to hide unrelated fine-grained differences when needed, streamlining the focus on resolution. Additionally, the layering mechanism (section 6) enables the hiding of semantic differences, conflicts, and resolutions, ensuring that consolidated changes adhere to DSL semantics.

10.3. Automatic Resolution of Conflicts

An ongoing challenge is the scarcity of existing conflict resolution techniques and tools capable of automatically addressing all conflict situations. While certain approaches (Hachemi and Ahmed-Nacer, 2020; Fritsche et al, 2020) perform resolution for limited conflict scenarios, they remain constrained. To enhance this, the domain-specific conflict resolution rule editor of DSMCompare empowers users to define conflict resolution rules including conflict specifications and relevant resolution patterns. DSL engineers can expand the repository of rules and tailor existing conflict specifications and resolution components. These rules are then translated into equivalent Henshin rules (Strüber et al, 2017), enabling their utilization by external tools, such as machine learning applications, for enhanced effectiveness (Eisenberg et al, 2021).

Some methodologies (e.g., (Wieland et al, 2013) and (Tröls et al, 2019)) fully support manual conflict resolution, involving users in making the final choices among suggestions during the resolution phase. However, a lack of insight into the intentions each user had during modeling can result in overlooked support for established requirements or even give rise to new conflicts (Chong et al, 2016; Brosch et al, 2012d). To aid DSL users in proper conflict resolution, DSMCompare offers diverse options explained in Section 7. Moreover, we provide enriched domain-specific and semantical information about the conflicts. For instance, utilizing domain-specific concrete syntax, we pinpoint the origin of conflicts. With appropriate styling, we denote the nature of conflicts. In semantic conflicts, users can address multiple fine-grained conflicts related to a single semantic conflict by making just one decision. This avoids incompatible choices for the fine-grained differences that cause semantic conflict, ensuring the original intention is preserved.

10.4. Support of Ordered Features

An ongoing research challenge pertains to the incorporation of ordered features, such as arrays of values or associations, into conflict management activities (Sharbaf et al, 2022b). In such cases, the ordering property assigns absolute indices to each value in collections and multi-valued features. However, merging two versions can lead to conflicts due to differing indices for features with identical values. Only three approaches, including (Koshima and Englebert, 2015; Dam et al, 2016; Schröpfer et al, 2019), have been fully implemented the ordered features, underscoring the necessity for further empirical research (Sharbaf et al, 2022b).

Our `DSDiffMM` metamodel is designed to encompass both ordered and unordered multi-valued features.This capability allows us to track changes in multi-valued features. As detailed in section 3, DSMCompare automatically generates the Conflict Resolution Rule metamodel (*CRRuleMM*) from the `DSDiffMM` metamodel. Consequently, DSL engineers are empowered to create conflict resolution rules based on changes to multi-valued elements.

11. Conclusion

This paper introduces an approach for domain-specific conflict resolution and model merging based on a three-way comparison. Our solution is integrated into a new version of DSMCompare that previously handled two-way and three-way domain-specific differencing and conflict detection. It automatically generates a domain-specific editor to create conflict resolution rules and enhances the concrete syntax to allow DSL users to visualize the three-way conflict resolutions more effectively. It supports merging trivial changes and resolving semantic conflicts when a predefined conflict resolution rule is available. This solution enables DSL users to manage conflicts in an environment familiar to their DSL, navigate between conflicts, manually resolve conflicts that need user intervention, undo previous resolution decisions, and save partial resolutions.

We evaluated our approach on multiple open-source projects. The results demonstrate that DSMCompare is highly effective at resolving fine-grained and semantic conflicts with a high degree of accuracy. The dataset of model versions involved in the commit history of several open-source projects, along with their labeled fine-grained and semantic conflicts and resolutions, is also available for future research.

We plan to incorporate a conflict reconciliation mechanism that leverages artificial intelligence techniques to learn implicit user preferences, such as in (Sharbaf et al, 2022a). This will lead to a final merged model free of conflicts that can be committed to a VCS repository. Additionally, we aim to integrate DSMCompare into domain-specific VCS systems using web-based editors. Another avenue of future research is the investigation of consistency checking after applying automatic conflict resolution.

Chapter 7

Conclusion

The main objective of VCS is to track and manage changes in project assets for effective collaboration and version control.

The primary objective of this thesis is to focus on the domain-specific aspects of differencing and merging within the context of MDE.

We focused on three main aspects including *enhancing semantic differencing and visualization*, *effective detection and visualization of semantic conflicts* and *empowering conflict resolution in domain-specific contexts.*

1. Summary

We summarize our three main contributions.

1.1. Enhancing semantic differencing and visualization

We have presented a comprehensive approach aimed at representing domain-specific model differences across both abstract and concrete syntax levels. Our methodology relies on automated modifications to the DSL metamodel, enabling the representation of fine-grained differences. Additionally, it involves the specification of semantic differencing rules, which capture recurring changes and are generated automatically through an editor. Furthermore, our approach includes the graphical representation of changes utilizing the DSL's syntax, achieved by automatically adjusting the DSL's concrete syntax specification. The practical implementation of our approach is realized through the DSMCompare tool, which seamlessly integrates into the Eclipse Modeling Framework and effectively handles graphical concrete syntaxes specified with Sirius.

Our approach's practicality has been validated through extensive experience with multiple case studies, including the configuration of the Pacman game, Arduino modeling, and metamodel refactoring, along with various experiments. These experiences have demonstrated the effectiveness of our approach in representing meaningful model differences in a domain-specific context. With DSMCompare, domain experts can visualize changes using the DSL's concrete syntax, allowing for the interpretation of semantically significant alterations within the domain. This approach results in concise, yet valuable differences that greatly benefit domain experts.

1.2. Effective detection and visualization of semantic conflicts

We have introduced an innovative approach focused on the detection of fine-grained and semantic differences and conflicts through a comprehensive three-way comparison framework. This advancement builds upon the previous version of DSMCompare, which primarily accommodated two-way domain-specific differences. The enhanced version now encompasses the capability to identify and represent both equivalent and contradicting conflicts that may arise between model versions.

DSMCompare empowers users to create semantic rules that automate the aggregation of fine-grained differences while attributing domain-specific significance to conflicts, streamlining the conflict resolution process. Additionally, we have refined the concrete syntax to provide DSL users with improved visualizations of three-way conflicts and differences, facilitating more effective comprehension.

Our approach's effectiveness was rigorously evaluated across various well-known open-source projects, yielding highly accurate results in the detection of semantic differences and conflicts. Furthermore, to facilitate future research endeavors, we have collected a substantial dataset comprising model versions from the commit history of several open-source projects, complete with labeled fine-grained and semantic differences and conflicts, which is now readily available for further exploration and analysis.

1.3. Empowering conflict resolution in domain-specific contexts

We have introduced an approach for domain-specific conflict resolution and model merging through a three-way comparison framework. This solution is seamlessly integrated into a new version of DSMCompare, which previously handled both two-way and three-way domain-specific differencing and conflict detection.

The enhanced version of DSMCompare now offers automatic generation of domain-specific editors for creating conflict resolution rules, alongside improvements in concrete syntax to facilitate more effective visualization of three-way conflict resolutions for DSL users. This solution is designed to handle tasks like merging trivial changes and resolving semantic conflicts when predefined conflict resolution rules are available.

Our approach empowers DSL users to efficiently manage conflicts within their familiar DSL environment. Users can navigate between conflicts, manually resolve conflicts requiring user intervention, undo previous resolution decisions, and save partial resolutions.

Through extensive evaluations conducted on various well-known open-source projects, we have demonstrated that DSMCompare excels in resolving fine-grained and semantic conflicts with a remarkable level of accuracy. Furthermore, we have made available a comprehensive dataset comprising model versions from the commit history of several open-source projects, complete with labeled fine-grained and semantic conflicts and their corresponding resolutions, fostering opportunities for future research and exploration.

2. Limitations of DSMCompare and Potential Areas for Improvement

In this section, we provide an overview of the limitations of DSMCompare and highlight the main areas for enhancing its functionalities

2.1. Technical Tooling

- *Underlying Technologies:* DSMCompare relies exclusively on EMF and Sirius. This exclusive reliance limits interoperability with other modeling technologies, potentially hindering integration with diverse tools and systems.
- *Performance Concerns:* Background processing for conflict detection in SDRules (Henshin, (CPA)) is slow, despite fast runtime for features like diff and merge.
- *Java Dependency:* Developed in Java, DSMCompare has limited integration with related technologies to VCS. The tool's development in Java restricts integration

with various version control systems, potentially limiting its adaptability to different development environments.

2.2. Semantics Diffs

- *Limited Behavioral Detection:* DSMCompare only detects coarse-grained structural differences, not behavioral differences. The tool may not capture nuanced behavioral changes, potentially overlooking important aspects of model evolution.
- *Expressiveness of SDRules:* Model transformation patterns in SDRule do not work *For All* quantifier. They only work *Exists* plus constraints.
- *Rule Creation Process:* Manual creation of the rules is tedious, although DSMCompare tries to streamline it with an example-based rule definition approach.
- *Dependency on Rules:* DSMCompare highly depends on the availability of rules; without them, it may not identify semantic differences or conflicts even if they exist.

2.3. Visualization

- *Graphical Syntax Limitation:* DSMCompare supports only graphical concrete syntax. Limited support for non-graphical syntax may constrain users who prefer alternative representation methods.
- *Scalability Challenges:* It does not scale well to very large models, despite attempts to address it with techniques like Juxtaposition, explicit encoding, superposition, and multiple layers.
- *Resolution Consequence Understanding:* The tool needs more UI mechanisms to help users understand the consequences of resolution decisions. It may lead to uncertainties in decision-making during conflict resolution.

2.4. Metamodel

- *Local Resolution Impact:* Local resolution by a DSL user may create conflicts or an invalid model, and DSMCompare cannot detect if it invalidates the system as a whole. Local resolutions may inadvertently introduce conflicts or invalidate the entire model, potentially impacting overall model consistency.
- *Metamodel Evolution:* Metamodel evolution does not automatically coevolve generated artifacts in DSMCompare. Changes in the metamodel may not automatically

propagate to existing artifacts, leading to inconsistencies between the metamodel and generated models.

- *Static Semantics Constraints:* Constraints in static semantics (OCL) are not supported. The absence of support for static semantics constraints may limit the tool's ability to enforce certain modeling constraints, potentially leading to non-compliance.
- *Well-Formedness Guarantee:* DSMCompare cannot guarantee that a merged model is well-formed with respect to the metamodel, although it depends on EMF APIs to check well-formedness.

3. Future Outlook of DSMCompare

In this section, we present a future outlook for DSMCompare.

Current Development and Integration Efforts: Currently, we are conducting a controlled experiment with users to assess the practical usability and performance of DSMCompare in real-world scenarios. The study aims to address crucial questions, such as whether domain-specific differencing and merging enhance usability compared to generic approaches.

Applications in DSM: DSMCompare lays the foundation for advancing domain-specific modeling by providing a robust platform for semantic differencing, conflict detection, and conflict resolution (Sharbaf et al, 2022b). Its future applications could extend to a broader range of DSM scenarios, contributing to enhanced model evolution and version control across diverse domains.

Configuration Management Systems: DSMCompare's capabilities align with the needs of configuration management systems. Future integration with these systems can streamline versioning processes and enhance collaboration in software development (Mehmood et al, 2020). The tool's proficiency in handling fine-grained differences and semantic conflicts can contribute to the efficiency and reliability of configuration management in complex projects.

Advancements in Semantic Conflict Management: The success of DSMCompare in empowering domain experts to define and manage semantic conflicts hints at future enhancements. Potential developments may include more sophisticated conflict pattern languages, and advanced visualization techniques in domain-specific languages (Reiter et al, 2007).

Extended Support for Multi-Domain Modeling: DSMCompare's flexibility in supporting various DSLs positions it for future extensions into multi-domain modeling scenarios (Liu et al, 2011). The tool could evolve to handle cross-domain differences, conflicts, and resolutions, catering to the growing complexity of modern software systems that often involve diverse modeling languages.

Enhanced Visualization Techniques: The configurable visualization approach of DSMCompare sets the stage for future advancements in graphical representations (Rautek et al, 2014). Enhanced visualizations, such as 3D modeling of conflicts, animated visual cues for semantic changes, and immersive interfaces, could provide users with more intuitive and informative views of model differences and conflicts.

Auditing Potential in DSMCompare: DSMCompare, with its robust conflict resolution framework tailored for domain-specific contexts, holds significant potential for integrating auditing capabilities into the model evolution process (CDO Model repository, accessed August 2023). Future developments could focus on enhancing audit trails, ensuring traceability of conflict resolution decisions, and exploring integrations with external auditing tools. By automating auditing processes, visualizing audit trails, and offering customizable workflows, DSMCompare could become a valuable asset for industries where compliance and auditing are critical components of model management. The tool's foundation in domain-specific conflict resolution positions it as a promising solution for maintaining transparency, accountability, and adherence to compliance standards throughout the model lifecycle.

Machine Learning and AI Integration: The wealth of labeled data in the dataset compiled by DSMCompare opens avenues for leveraging machine learning and artificial intelligence techniques. Future developments may explore the integration of intelligent algorithms for automated conflict resolution suggestions, adaptive conflict pattern recognition, and personalized conflict resolution strategies based on historical user interactions (Sharbaf et al, 2022a). Furthermore, ensuring the quality and integrity of the final merged model is another challenge, requiring consistency checking after automatic conflict resolution.

Bibliography

[flake8 2023] *Flake8*. `https://flake8.pycqa.org/en/latest/`. 2023. – (accessed August 2023)

[githubActions 2023] *Github Actions*. `https://docs.github.com/en/actions`. 2023. – (accessed August 2023)

[gitlabCICD 2023] *GitLab CI/CD*. `https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/`. 2023. – (accessed August 2023)

[UML 2023] *UNIFIED MODELING LANGUAGE*. `[https://www.omg.org/spec/UML/About-UML/]`. 2023. – [accessed August 2023]

[van der Aalst 1998] Aalst, Wil M P. van der: The Application of Petri nets to Workflow Management. In: *Journal of Circuits, Systems and Computers* 8 (1998), Nr. 1, p. 21–66

[Addazi et al 2016] Addazi, Lorenzo ; Cicchetti, Antonio ; Di Rocco, Juri ; Di Ruscio, Davide ; Iovino, Ludovico ; Pierantonio, Alfonso: Semantic-based Model Matching with EMFCompare. In: *Workshop on Models and Evolution* Volume 1706, CEUR-WS.org, 2016, p. 40–49

[Adipat et al 2011] Adipat, Boonlit ; Zhang, Dongsong ; Zhou, Lina: The effects of tree-view based presentation adaptation on mobile web browsing. In: *Mis Quarterly* (2011), p. 99–121

[Akdur et al 2018] Akdur, Deniz ; Garousi, Vahid ; Demirörs, Onur: A survey on modeling and model-driven engineering practices in the embedded software industry. In: *Journal of Systems Architecture* 91 (2018), p. 62–82

[Al-Herz and Pothen 2019] Al-Herz, Ahmed ; Pothen, Alex: A 2/3-approximation algorithm for vertex-weighted matching. In: *Discrete Applied Mathematics* (2019), p. 46–67

[Altmanninger et al 2008a] Altmanninger, K. ; Kappel, G. ; Kusel, A. ; Retschitzegger, W. ; Seidl, M. ; Schwinger, W. ; Wimmer, M.: AMOR - Towards Adaptable Model Versioning. In: *1st International Workshop on Model Co-Evolution and Consistency Management*, 2008, p. 4–50. – See also `http://www.modelversioning.org/`

[Altmanninger and Pierantonio 2011] Altmanninger, K ; Pierantonio, Alfonso: A categorization for conflicts in model versioning. In: *e & i Elektrotechnik und Informationstechnik* 128 (2011), Nr. 11-12, p. 421–426

[Altmanninger 2007] Altmanninger, Kerstin: Models in conflict–towards a semantically enhanced version control system for models. In: *International Conference on Model Driven Engineering Languages and Systems* Springer (event), 2007, p. 293–304

[Altmanninger et al 2008b] Altmanninger, Kerstin ; Kappel, Gerti ; Kusel, Angelika ; Retschitzegger, Werner ; Seidl, Martina ; Schwinger, Wieland ; Wimmer, Manuel: AMOR–towards adaptable model versioning. In: *Workshop on Model Co-Evolution and Consistency Management* Volume 8, 2008, p. 4–50

[Altmanninger et al 2010] Altmanninger, Kerstin ; Schwinger, Wieland ; Kotsis, Gabriele: Semantics for accurate conflict detection in smover: Specification, detection and presentation by example. In: *International Journal of Enterprise Information Systems (IJEIS)* 6 (2010), Nr. 1, p. 68–84

[Altmanninger et al 2009] Altmanninger, Kerstin ; Seidl, Martina ; Wimmer, Manuel: A survey on model versioning approaches. In: *International Journal on Web Information Systems* 5 (2009), Nr. 3, p. 271–304

[Arendt et al 2010] Arendt, Thorsten ; Biermann, Enrico ; Jurack, Stefan ; Krause, Christian ; Taentzer, Gabriele: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: *Model Driven Engineering Languages and Systems* Volume 6394, Springer, 2010, p. 121–135

[Önder Babur et al 2019] Babur Önder ; Cleophas, Loek ; van den Brand, Mark: Metamodel clone detection with SAMOS. In: *Journal of Computer Languages* 51 (2019), p. 57–74. – ISSN 2590-1184

[Baqasah et al 2014] Baqasah, Abdullah ; Pardede, Eric ; Rahayu, Wenny: A new approach for meaningful XML schema merging. In: *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services*, 2014, p. 430–439

[Barrett et al 2011] Barrett, Stephen C. ; Chalin, Patrice ; Butler, Greg: Table-driven detection and resolution of operation-based merge conflicts with mirador. In: *Modelling Foundations and Applications: 7th European Conference, ECMFA 2011, Birmingham, UK, June 6-9, 2011 Proceedings 7* Springer (event), 2011, p. 329–344

[Bartelt 2008] Bartelt, Christian: Consistence preserving model merge in collaborative development processes. In: *Proceedings of the 2008 international workshop on Comparison and versioning of software models*, 2008, p. 13–18

[Bartelt and Schindler 2010] Bartelt, Christian ; Schindler, Bjorn: Technology support for collaborative inconsistency management in model driven engineering. In: *2010 43rd Hawaii International Conference on System Sciences* IEEE (event), 2010, p. 1–7

[Basciani et al 2014] Basciani, F. ; Rocco, J. D. ; Ruscio, D. D. ; Salle, A. D. ; Iovino, L. ; Pierantonio, A.: MDEForge: an extensible web-based modeling platform. In: *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud*, CEUR-WS.org, 2014, p. 66–75

[Berkenkötter 2008] Berkenkötter, Kirsten: Reliable UML Models and Profiles. In: *Electronic Notes in Theoretical Computer Science* 217 (2008), p. 203–220

[Bernstein 2003] Bernstein, Philip A.: Applying Model Management to Classical Meta Data Problems. In: *CIDR* Volume 2003, 2003, p. 209–220

[Biermann et al 2012] Biermann, Enrico ; Ermel, Claudia ; Taentzer, Gabriele: Formal foundation of consistent EMF model transformations by algebraic graph transformation. In: *Software & Systems Modeling* 11 (2012), Nr. 2, p. 227–250

[Blanc et al 2008] Blanc, Xavier ; Mounier, Isabelle ; Mougenot, Alix ; Mens, Tom: Detecting model inconsistency through operation-based model construction. In: *Proceedings of the 30th international conference on Software engineering*, 2008, p. 511–520

[Booch et al 2000] Booch, Grady ; Jacobson, Ivar ; Rumbaugh, Jim: OMG unified modeling language specification. In: *Object Management Group* 1034 (2000), p. 15–44

[Brambilla et al 2012] Brambilla, Marco ; Cabot, Jordi ; Wimmer, Manuel: *Model-Driven Software Engineering in Practice.* Morgan & Claypool Publishers, 2012

[van den Brand et al 2010] Brand, M. van den ; Protić, Z. ; Verhoeff, T.: Generic Tool for Visualization of Model Differences. In: *Workshop on Model Comparison in Practice*, ACM, 2010, p. 66–75

[Brindescu et al 2020] Brindescu, Caius ; Ahmed, Iftekhar ; Jensen, Carlos ; Sarma, Anita: An empirical investigation into merge conflicts and their effect on software quality. In: *Empirical Software Engineering* 25 (2020), p. 562–590

[Brosch et al 2012a] Brosch, P. ; Kappel, G. ; Langer, P. ; Seidl, M. ; Wieland, K. ; Wimmer, M.: An introduction to model versioning. In: *SFM* Volume 7320, Springer, 2012, p. 336–398

[Brosch et al 2009] Brosch, P. ; Seidl, M. ; Wieland, K. ; Wimmer, M.: We can work it out: Collaborative conflict resolution in model versioning. In: *European Conference on Computer-Supported Cooperative Work*, Springer, 2009, p. 207–214

[Brosch et al 2012b] Brosch, P. ; Seidl, M. ; Wimmer, M. ; Kappel, G.: Conflict visualization for evolving UML models. In: *Journal of Object Technology* 11 (2012), Nr. 3, p. 2:1–30

[Brosch et al 2012c] Brosch, Petra ; Egly, Uwe ; Gabmeyer, Sebastian ; Kappel, Gerti ; Seidl, Martina ; Tompits, Hans ; Widl, Magdalena ; Wimmer, Manuel: Towards Semantics-Aware Merge Support in Optimistic Model Versioning. In: *Models in Software Engineering.* Springer, 2012, p. 246–256

[Brosch et al 2012d] Brosch, Petra ; Kappel, Gerti ; Langer, Philip ; Seidl, Martina ; Wieland, Konrad ; Wimmer, Manuel: An introduction to model versioning. In: *International school on formal methods for the design of computer, communication and software systems.* Springer, 2012, p. 336–398

[Brosch et al 2010a] Brosch, Petra ; Kappel, Gerti ; Seidl, Martina ; Wieland, Konrad ; Wimmer, Manuel ; Kargl, Horst ; Langer, Philip: Adaptable Model Versioning in Action. In: *Modellierung* Volume 161, GI, 2010, p. 24–26

[Brosch et al 2010b] Brosch, Petra ; Kappel, Gerti ; Seidl, Martina ; Wieland, Konrad ; Wimmer, Manuel ; Kargl, Horst ; Langer, Philip: Adaptable model versioning in action. In: *Modellierung 2010* (2010)

[Brosch et al 2010c] Brosch, Petra ; Langer, Philip ; Seidl, Martina ; Wieland, Konrad ; Wimmer, Manuel: Colex: a web-based collaborative conflict lexicon. In: *Proceedings of the 1st International Workshop on Model Comparison in Practice*, 2010, p. 42–49

[Brosch et al 2012e] Brosch, Petra ; Langer, Philip ; Seidl, Martina ; Wieland, Konrad ; Wimmer, Manuel ; Kappel, Gerti: The past, present, and future of model versioning. In: *Emerging Technologies for the Evolution and Maintenance of Software Models*. IGI Global, 2012, p. 410–443

[Brosch et al 2010d] Brosch, Petra ; Seidl, Martina ; Kappel, Gerti: A recommender for conflict resolution support in optimistic model versioning. In: *Proc. SPLASH/OOPSLA Companion*, ACM, 2010, p. 43–50

[Brosch et al 2012f] Brosch, Petra ; Seidl, Martina ; Wimmer, Manuel ; Kappel, Gerti: Conflict Visualization for Evolving UML Models. In: *J. Object Technol.* 11 (2012), Nr. 3, p. 2–1

[Brun and Pierantonio 2008] Brun, Cédric ; Pierantonio, Alfonso: Model differences in the Eclipse Modeling Framework. In: *UPGRADE, The European Journal for the Informatics Professional* 9 (2008), Nr. 2, p. 29–34. – `https://www.eclipse.org/emf/compare/`

[Bruneliere et al 2014] Bruneliere, Hugo ; Cabot, Jordi ; Dupé, Grégoire ; Madiot, Frédéric: Modisco: A model driven reverse engineering framework. In: *Information and Software Technology* 56 (2014), Nr. 8, p. 1012–1032

[CDO Model repository accessed August 2023]

[Chong et al 2016] Chong, Hao ; Zhang, Renwei ; Qin, Zheng: Composite-based conflict resolution in merging versions of UML models. In: *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)* IEEE (event), 2016, p. 127–132

[Cicchetti et al 2007] Cicchetti, A. ; Ruscio, D. D. ; Pierantonio, A.: A metamodel independent approach to difference representation. In: *Journal of Object Technology* 6 (2007), Nr. 9, p. 165–185

[Cicchetti et al 2008a] Cicchetti, Antonio ; Di Ruscio, Davide ; Pierantonio, Alfonso: Managing Model Conflicts in Distributed Development. In: Czarnecki, Krzysztof (Editor) ; Ober, Ileana (Editor) ; Bruel, Jean-Michel (Editor) ; Uhl, Axel (Editor) ; Völter, Markus (Editor): *Model Driven Engineering Languages and Systems*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 311–325. – URL `https://doi.org/10.1007/978-3-540-87875-9_23`. – ISBN 978-3-540-87875-9

[Cicchetti et al 2008b] Cicchetti, Antonio ; Di Ruscio, Davide ; Pierantonio, Alfonso: Managing model conflicts in distributed development. In: *Model Driven Engineering Languages and Systems: 11th International Conference, MoDELS 2008, Toulouse, France, September 28-October 3, 2008. Proceedings 11* Springer (event), 2008, p. 311–325

[Cicchetti et al 2010] Cicchetti, Antonio ; Di Ruscio, Davide ; Pierantonio, Alfonso: 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* Springer (event), 2010, p. 190–204

[Dam et al 2016] Dam, Hoa K. ; Egyed, Alexander ; Winikoff, Michael ; Reder, Alexander ; Lopez-Herrejon, Roberto E.: Consistent merging of model versions. In: *Journal of Systems and Software* 112 (2016), p. 137–155

[David et al 2021] David, Istvan ; Aslam, Kousar ; Faridmoayer, Sogol ; Malavolta, Ivano ; Syriani, Eugene ; Lago, Patricia: Collaborative Model-Driven Software Engineering: A Systematic Update. In: *Model Driven Engineering Languages and Systems*, ACM, 2021, p. 273–284

[Debreceni et al 2016] Debreceni, Csaba ; Ráth, István ; Varró, Dániel ; De Carlos, Xabier ; Mendialdua, Xabier ; Trujillo, Salvador: Automated model merge by design space exploration. In: *International Conference on Fundamental Approaches to Software Engineering* Springer (event), 2016, p. 104–121

[DiffMerge 2023] DiffMerge, EMF: `https://wiki.eclipse.org/EMF_DiffMerge`. 2023. – last accessed 2023

[Dig et al 2007] Dig, Danny ; Manzoor, Kashif ; Johnson, Ralph ; Nguyen, Tien N.: Refactoring-Aware Configuration Management for Object-Oriented Programs. In: *Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2007 (ICSE '07), p. 427–436. – URL `https://doi.org/10.1109/ICSE.2007.71`. – ISBN 0-7695-2828-7

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

[Eclipse Foundation 2023a] Eclipse Foundation: *ATL*. `https://www.eclipse.org/atl/`. 2023. – Last accessed on: November 28, 2023

[Eclipse Foundation 2023b] Eclipse Foundation: *Eclipse Modeling Framework (EMF).* `https://www.eclipse.org/modeling/emf/docs/`. 2023. – Last accessed on: November 28, 2023

[Eclipse Foundation 2023c] Eclipse Foundation: *Epsilon Generation Language (EGL).* `https://www.eclipse.org/epsilon/doc/egl/`. 2023. – Last accessed on: November 28, 2023

[Ehrig et al 2006] Ehrig, H. ; Ehrig, K. ; Prange, U. ; Taentzer, G.: *Fundamentals of Algebraic Graph Transformation.* Springer, 2006 (Monographs in Theoretical Computer Science. An EATCS Series)

[Eisenberg et al 2021] Eisenberg, Martin ; Pichler, Hans-Peter ; Garmendia, Antonio ; Wimmer, Manuel: Towards reinforcement learning for in-place model transformations. In: *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)* IEEE (event), 2021, p. 82–88

[Ellis et al 2023] Ellis, Max ; Nadi, Sarah ; Dig, Danny: Operation-Based Refactoring-Aware Merging: An Empirical Evaluation. In: *IEEE Transactions on Software Engineering* 49 (2023), Nr. 4, p. 2698–2721

[EMF Compare accessed August 2023]

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

[epsilon 2023] epsilon: `http://www.eclipse.org/epsilon/`. 2023. – (accessed August 2023)

[Espinazo-Pagán et al 2011] Espinazo-Pagán, Javier ; Cuadrado, Jesús S. ; Molina, Jesús García: Morsa: A Scalable Approach for Persisting and Accessing Large Models. In: *Model-Driven Language Engineering and Systems* Volume 6981, Springer, 2011, p. 77–92

[Espinazo-Pagán and García-Molina 2010] Espinazo-Pagán, Javier ; García-Molina, Jesús: A Homogeneous Repository for Collaborative MDE. In: *Workshop on Model Comparison in Practice*, ACM, 2010 (IWMCP), p. 56–65

[Frankel 2002] Frankel, David S.: *Model-Driven Architecture: Applying MDA to Enterprise Computing.* John Wiley & Sons, 2002

[Franzago et al 2017] Franzago, Mirco ; Di Ruscio, Davide ; Malavolta, Ivano ; Muccini, Henry: Collaborative model-driven software engineering: a classification framework and a research map. In: *IEEE Transactions on Software Engineering* 44 (2017), Nr. 12, p. 1146–1175

[Fritsche et al 2020] FRITSCHE, Lars ; KOSIOL, Jens ; MÖLLER, Adrian ; SCHÜRR, Andy ; TAENTZER, Gabriele: A precedence-driven approach for concurrent model synchronization scenarios using triple graph grammars. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, 2020, p. 39–55

[García et al 2013] GARCÍA, J. ; DIAZ, O. ; AZANZA, M.: Model transformation co-evolution: A semi-automatic approach. In: *SLE* Volume 7745, Springer, 2013, p. 144–163

[Git last accessed 2023]

[Gleicher 2018] GLEICHER, M.: Considerations for visualizing comparison. In: *Transactions on Visualization and Computer Graphics* 24 (2018), Nr. 1, p. 413–423

[Gleicher et al 2011] GLEICHER, Michael ; ALBERS, Danielle ; WALKER, Rick ; JUSUFI, Ilir ; HANSEN, Charles D. ; ROBERTS, Jonathan C.: Visual comparison for information visualization. In: *Information Visualization* 10 (2011), Nr. 4, p. 289–309

[GMF last accessed 2023]

[Hachemi and Ahmed-Nacer 2020] HACHEMI, Asma ; AHMED-NACER, Mohamed: Conflict resolution in process models merging. In: *Software Engineering Perspectives in Intelligent Systems: Proceedings of 4th Computational Methods in Systems and Software 2020, Vol. 1 4* Springer (event), 2020, p. 336–345

[Haeusler et al 2019] HAEUSLER, Martin ; TROJER, Thomas ; KESSLER, Johannes ; FARWICK, Matthias ; NOWAKOWSKI, Emmanuel ; BREU, Ruth: ChronoSphere: a graph-based EMF model repository for IT landscape models. In: *Software and Systems Modeling* 18 (2019), Nr. 6, p. 3487–3526

[Herrmannsdoerfer et al 2009] HERRMANNSDOERFER, Markus ; RATIU, Daniel ; WACHSMUTH, Guido: Language evolution in practice: The history of GMF. In: *International Conference on Software Language Engineering* Springer (event), Springer, 2009, p. 3–22

[Holthusen et al 2014] HOLTHUSEN, Sönke ; WILLE, David ; LEGAT, Christoph ; BEDDIG, Simon ; SCHAEFER, Ina ; VOGEL-HEUSER, Birgit: Family model mining for function block diagrams in automation software. In: *Proc SPLC Companion*, ACM, 2014, p. 36–43

[Jackson and Ladd 1994] JACKSON, Daniel ; LADD, David A.: Semantic Diff: a tool for summarizing the effects of modifications. In: *International Conference on Software Maintenance*, IEEE, 1994, p. 243–252

[Jouault et al 2008] JOUAULT, Frédéric ; ALLILAIRE, Freddy ; BÉZIVIN, Jean ; KURTEV, Ivan: ATL: A model transformation tool. In: *Science of computer programming* 72 (2008), Nr. 1-2, p. 31–39

[Kappel et al 2006] KAPPEL, Gerti ; KAPSAMMER, Elisabeth ; KRAMLER, Gerhard ; REITER, Thomas ; RETSCHITZEGGER, Werner ; SCHWINGER, Wieland: Towards A Semantic Infrastructure Supporting Model-based Tool Integration. In: *Proc. GaMMa'06*, ACM, 2006 (GaMMa '06), p. 43–46

[Kaufmann et al 2010] KAUFMANN, Petra ; KAPPEL, Gerti ; SEIDL, Martina ; WIELAND, Konrad ; WIMMER, Manuel ; KARGL, Horst ; LANGER, Philip: Adaptable Model Versioning in Action. In: *Modellierung 2010* GI (event), 2010, p. 221–236

[Kautz and Rumpe 2018] KAUTZ, Oliver ; RUMPE, Bernhard: On computing instructions to repair failed model refinements. In: *Model Driven Engineering Languages and Systems*, 2018, p. 289–299

[kdiff3 2023] KDIFF3: `https://kdiff3.net/`. 2023. – (accessed August 2023)

[Kehrer et al 2011] KEHRER, T. ; KELTER, U. ; TAENTZER, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: *Automated Software Engineering*, IEEE Computer Society, 2011, p. 163–172

[Kehrer et al 2013] KEHRER, Timo ; KELTER, Udo ; TAENTZER, Gabriele: Consistency-preserving edit scripts in model versioning. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* IEEE (event), 2013, p. 191–201

[Kelly and Tolvanen 2008] KELLY, S. ; TOLVANEN, J-K.: *Domain-Specific Modeling - Enabling Full Code Generation.* Wiley, 2008

[Kelly 2018] KELLY, Steven: Collaborative Modelling with Version Control. In: *Software Technologies: Applications and Foundations* Volume 10748, Springer, 2018, p. 20–29

[Kelly et al 1996] KELLY, Steven ; LYYTINEN, Kalle ; ROSSI, Matti: MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In: IIVARI, Juhani (Editor) ; LYYTINEN, Kalle (Editor) ; ROSSI, Matti (Editor): *Conference on Advanced Information Systems Engineering* Volume 1080. Crete : Springer-Verlag, 6 1996, p. 1–21

[Kelly and Tolvanen 2008] Kelly, Steven ; Tolvanen, Juha-Pekka: *Domain-Specific Modeling: Enabling Full Code Generation.* John Wiley & Sons, 2008

[Kent 2002] Kent, Stuart: Model driven engineering. In: *International conference on integrated formal methods* Springer (event), 2002, p. 286–298

[Koegel and Helming 2010a] Koegel, Maximilian ; Helming, Jonas: EMFStore: a model repository for EMF models. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2* Volume 2 ACM (event), ACM, 2010, p. 307–308

[Koegel and Helming 2010b] Koegel, Maximilian ; Helming, Jonas: EMFStore: a model repository for EMF models. In: *Proceedings ICSE Vol 2*, ACM, 2010, p. 307–308. – See also `https://www.eclipse.org/emfstore/`

[Koegel and Langer 2015] Koegel, Maximilian ; Langer, Philip: Integrating open-source modeling projects: Collaborative modeling with Papyrus and EMF Compare. In: *Proceedings of the International Workshop on Open Source Software for Model Driven Engineering*, 2015

[Kolovos 2009] Kolovos, Dimitrios S.: 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* Springer (event), 2009, p. 146–157

[Kolovos et al 2009a] Kolovos, Dimitrios S. ; Di Ruscio, Davide ; Pierantonio, Alfonso ; Paige, Richard F.: Different Models for Model Matching: An analysis of approaches to support model differencing. In: *Comparison and Versioning of Software Models* IEEE (event), IEEE, 2009, p. 1–6

[Kolovos et al 2006a] Kolovos, Dimitrios S. ; Paige, Richard F. ; Polack, Fiona A.: The epsilon object language (EOL). In: *European conference on model driven architecture-foundations and applications* Springer (event), 2006, p. 128–142

[Kolovos et al 2006b] Kolovos, Dimitrios S. ; Paige, Richard F. ; Polack, Fiona A.: Merging models with the epsilon merging language (eml). In: *International Conference on Model Driven Engineering Languages and Systems* Springer (event), 2006, p. 215–229

[Kolovos et al 2009b] Kolovos, Dimitrios S. ; Paige, Richard F. ; Polack, Fiona A.: On the evolution of OCL for capturing structural constraints in modelling languages. In: *Rigorous Methods for Software Construction and Analysis: Essays Dedicated to Egon*

Börger on the Occasion of His 60th Birthday (2009), p. 204–218

[Koshima and Englebert 2015] KOSHIMA, Amanuel A. ; ENGLEBERT, Vincent: Collaborative editing of EMF/Ecore meta-models and models: Conflict detection, reconciliation, and merging in DiCoMEF. In: *Science of Computer Programming* 113 (2015), p. 3–28

[Kühne et al 2009] KÜHNE, T. ; MEZEI, G. ; SYRIANI, E. ; VANGHELUWE, H. ; WIMMER, M.: Explicit transformation modeling. In: *MODELS 2009 Workshops* Volume 6002, Springer, 2009, p. 240–255

[Kuiter et al 2021] KUITER, Elias ; KRIETER, Sebastian ; KRÜGER, Jacob ; SAAKE, Gunter ; LEICH, Thomas: variED: an editor for collaborative, real-time feature modeling. In: *Empirical Software Engineering* 26 (2021), p. 1–47

[LabView 2023] LABVIEW: `https://www.ni.com/en-us/support/documentation/supplemental/21/managing-labview-vi-and-application-revision-history.html`. 2023. – (last accessed in March 2023)

[Lambers et al 2019] LAMBERS, Leen ; BORN, Kristopher ; KOSIOL, Jens ; STRÜBER, Daniel ; TAENTZER, Gabriele: Granularity of conflicts and dependencies in graph transformation systems: a two-dimensional approach. In: *Journal of logical and algebraic methods in programming* 103 (2019), p. 105–129

[Lambers et al 2006] LAMBERS, Leen ; EHRIG, Hartmut ; OREJAS, Fernando: Conflict Detection for Graph Transformation with Negative Application Conditions. In: *Proc. ICGT* Volume 4178, Springer, 2006, p. 61–76

[Lambers et al 2008] LAMBERS, Leen ; EHRIG, Hartmut ; OREJAS, Fernando: Efficient Conflict Detection in Graph Transformation Systems by Essential Critical Pairs. In: *Electronic Notes in Theoretical Computer Science* 211 (2008), p. 17–26

[Lambers et al 2018] LAMBERS, Leen ; STRÜBER, Daniel ; TAENTZER, Gabriele ; BORN, Kristopher ; HUEBERT, Jevgenij: Multi-granular conflict and dependency analysis in software engineering based on graph transformation. In: *International Conference on Software Engineering*, 2018, p. 716–727

[Langer et al 2012] LANGER, P. ; WIMMER, M. ; GRAY, J. ; KAPPEL, G. ; VALLECILLO, A.: Language-specific model versioning based on signifiers. In: *Journal of Object Technology* 11 (2012), Nr. 3, p. 4: 1–34

[Langer et al 2013a] Langer, Philip ; Wimmer, Manuel ; Brosch, Petra ; Herrmannsdörfer, Markus ; Seidl, Martina ; Wieland, Konrad ; Kappel, Gerti: A posteriori operation detection in evolving software models. In: *Journal of Systems and Software* 86 (2013), Nr. 2, p. 551–566

[Langer et al 2013b] Langer, Philip ; Wimmer, Manuel ; Brosch, Petra ; Herrmannsdörfer, Markus ; Seidl, Martina ; Wieland, Konrad ; Kappel, Gerti: A posteriori operation detection in evolving software models. In: *Journal of Systems and Software* 86 (2013), Nr. 2, p. 551 – 566. – URL `http://www.sciencedirect.com/science/article/pii/S0164121212002762`. – ISSN 0164-1212

[de Lara et al 2007] Lara, J. de ; Bardohl, R. ; Ehrig, H. ; Ehrig, K. ; Prange, U. ; Taentzer, G.: Attributed graph transformation with node type inheritance. In: *Theor. Comput. Sci.* 376 (2007), Nr. 3, p. 139–163

[Leßenich et al 2018] Lessenich, Olaf ; Siegmund, Janet ; Apel, Sven ; Kästner, Christian ; Hunsen, Claus: Indicators for merge conflicts in the wild: survey and empirical study. In: *Automated Software Engineering* 25 (2018), Nr. 2, p. 279–313

[Lin et al 2007] Lin, Y. ; Gray, J. ; Jouault, F.: DSMDiff: a differentiation tool for domain-specific models. In: *European Journal of Information Systems* 16 (2007), Nr. 4, p. 349–361

[Liu et al 2011] Liu, Yang ; Sun, Jun ; Dong, Jin S.: Pat 3: An extensible architecture for building multi-domain model checkers. In: *2011 IEEE 22nd international symposium on software reliability engineering* IEEE (event), 2011, p. 190–199

[Lúcio et al 2013] Lúcio, Levi ; Mustafiz, Sadaf ; Denil, Joachim ; Vangheluwe, Hans ; Jukss, Maris: FTG+ PM: An integrated framework for investigating model transformation chains. In: *SDL 2013: Model-Driven Dependability Engineering: 16th International SDL Forum, Montreal, Canada, June 26-28, 2013. Proceedings 16* Springer (event), 2013, p. 182–202

[Mahmoudi et al 2019] Mahmoudi, Mehran ; Nadi, Sarah ; Tsantalis, Nikolaos: Are refactorings to blame? an empirical study of refactorings in merge conflicts. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* IEEE (event), IEEE, 2019, p. 151–162

[Mannadiar 2012] Mannadiar, Raphaël: A multi-paradigm modelling approach to the foundations of domain-specific modelling. (2012)

[Mansoor et al 2015] Mansoor, Usman ; Kessentini, Marouane ; Langer, Philip ; Wimmer, Manuel ; Bechikh, Slim ; Deb, Kalyanmoy: MOMM: Multi-objective model merging. In: *Journal of Systems and Software* 103 (2015), may, p. 423–439

[Maoz and Ringert 2018] Maoz, S. ; Ringert, J. O.: A framework for relating syntactic and semantic model differences. In: *Software & System Modeling* 17 (2018), Nr. 3, p. 753–777

[Maoz et al 2011a] Maoz, S. ; Ringert, J. O. ; Rumpe, B.: A manifesto for semantic model differencing. In: *MODELS 2010 Workshops* Volume 6627, Springer, 2011, p. 194–203

[Maoz et al 2011b] Maoz, Shahar ; Ringert, Jan O. ; Rumpe, Bernhard: ADDiff: semantic differencing for activity diagrams. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, p. 179–189

[Mehmood et al 2020] Mehmood, Waqar ; Shafiq, Muhammad ; Saleem, Muhammad Q. ; Alowayr, Ali S. ; Aslam, Waqar: A Feature-Based Evaluation of Model Merge Methods for e-Health Solutions. In: *Journal of Medical Imaging and Health Informatics* 10 (2020), Nr. 10, p. 2473–2480

[Mehra et al 2005] Mehra, A. ; Grundy, J. C. ; Hosking, J. G.: A generic approach to supporting diagram differencing and merging for collaborative design. In: *Automated Software Engineering*, ACM, 2005, p. 204–213

[Mens 2002] Mens, Tom: A state-of-the-art survey on software merging. In: *IEEE transactions on software engineering* 28 (2002), Nr. 5, p. 449–462

[Mens et al 2005] Mens, Tom ; Taentzer, Gabriele ; Runge, Olga: Detecting structural refactoring conflicts using critical pair analysis. In: *Electronic Notes in Theoretical Computer Science* 127 (2005), Nr. 3, p. 113–128

[MetaEdit last accessed 2023]

[Mohagheghi et al 2013] Mohagheghi, Parastoo ; Gilani, Wasif ; Stefanescu, Alin ; Fernandez, Miguel A. ; Nordmoen, Bjørn ; Fritzsche, Mathias: Where Does Model-Driven Engineering Help? Experiences from Three Industrial Cases. 12 (2013), Nr. 3

[Mougenot et al 2009] MOUGENOT, Alix ; BLANC, Xavier ; GERVAIS, Marie-Pierre: D-praxis: A peer-to-peer collaborative model editing framework. In: *Distributed Applications and Interoperable Systems: 9th IFIP WG 6.1 International Conference, DAIS 2009, Lisbon, Portugal, June 9-11, 2009. Proceedings 9* Springer (event), 2009, p. 16–29

[MPS last accessed 2023a] MPS: *Differences viewer for files.* `https://www.jetbrains.com/help/mps/differences-viewer.html`. last accessed 2023

[MPS last accessed 2023b] MPS: *Version Control.* `https://www.jetbrains.com/help/mps/version-control-integration.html`. last accessed 2023

[Munaiah et al 2017] MUNAIAH, Nuthan ; KROH, Steven ; CABREY, Craig ; NAGAPPAN, Meiyappan: Curating github for engineered software projects. In: *Empirical Software Engineering* 22 (2017), Nr. 6, p. 3219–3253

[Nicolaescu et al 2018] NICOLAESCU, Petru ; ROSENSTENGEL, Mario ; DERNTL, Michael ; KLAMMA, Ralf ; JARKE, Matthias: Near real-time collaborative modeling for view-based web information systems engineering. In: *Information Systems* 74 (2018), p. 23–39

[Object Management Group (OMG) 2023] OBJECT MANAGEMENT GROUP (OMG): *The Object Constraint Language (OCL).* `https://www.omg.org/spec/OCL/2.4/About-OCL`. 2023. – Last accessed on: November 28, 2023

[Object Management Group (OMG) last accessed 2023] OBJECT MANAGEMENT GROUP (OMG): *The MetaObject Facility Specification(MOF).* `[https://www.omg.org/mof/]`. last accessed 2023

[Ohst et al 2003] OHST, D. ; WELLE, M. ; KELTER, U.: Differences between versions of UML diagrams. In: *ESEC/FSE*, ACM, 2003, p. 227–236

[OMG 2023] OMG: *XMI metadata interchange v. 2.5.1.* `https://www.omg.org/spec/XMI/About-XMI/`. 2023. – (last accessed in March 2023)

[Owhadi-Kareshk et al 2019] OWHADI-KARESHK, Moein ; NADI, Sarah ; RUBIN, Julia: Predicting merge conflicts in collaborative software development. In: *Symposium on Empirical Software Engineering and Measurement*, IEEE, 2019, p. 1–11

[Paige et al 2016] PAIGE, R. F. ; MATRAGKAS, N. D. ; ROSE, L. M.: Evolving models in Model-Driven Engineering: State-of-the-art and future challenges. In: *Journal of Systems and Software* 111 (2016), p. 272–280

[Pinto et al 2018] PINTO, Gustavo ; STEINMACHER, Igor ; DIAS, Luiz F. ; GEROSA, Marco: On the challenges of open-sourcing proprietary software projects. In: *Empirical Software Engineering* 23 (2018), Nr. 6, p. 3221–3247

[Rautek et al 2014] RAUTEK, Peter ; BRUCKNER, Stefan ; GRÖLLER, M E. ; HADWIGER, Markus: ViSlang: A system for interpreted domain-specific languages for scientific visualization. In: *IEEE Transactions on Visualization and Computer Graphics* 20 (2014), Nr. 12, p. 2388–2396

[Reiter et al 2007] REITER, Thomas ; ALTMANNINGER, Kerstin ; BERGMAYR, Alexander ; SCHWINGER, Wieland ; KOTSIS, Gabriele: Models in conflict-detection of semantic conflicts in model-based development. In: *MDEIS@ ICEIS* 7 (2007), p. 29–40

[Reuling et al 2019] REULING, Dennis ; LOCHAU, Malte ; KELTER, Udo: From Imprecise N-Way Model Matching to Precise N-Way Model Merging. In: *J. Object Technol.* 18 (2019), Nr. 2, p. 8:1–20

[Rivera and Vallecillo 2008] RIVERA, José E ; VALLECILLO, Antonio: 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* Springer (event), 2008, p. 141–160

[Rossini et al 2010] ROSSINI, Alessandro ; RUTLE, Adrian ; LAMO, Yngve ; WOLTER, Uwe: A formalisation of the copy-modify-merge approach to version control in MDE. In: *The Journal of Logic and Algebraic Programming* 79 (2010), Nr. 7, p. 636–658

[Rossini et al 2018] ROSSINI, Alessandro ; RUTLE, Adrian ; LAMO, Yngve ; WOLTER, Uwe E.: Handling constraints in model versioning. (2018)

[Rubin and Chechik 2013] RUBIN, Julia ; CHECHIK, Marsha: N-way model merging. In: *proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, p. 301–311

[Rumbaugh et al 1991] RUMBAUGH, James ; BLAHA, Michael ; PREMERLANI, William ; EDDY, Frederick ; LORENSEN, William E. et al: *Object-oriented modeling and design.* Prentice-hall Englewood Cliffs, NJ, 1991

[Sabetzadeh and Easterbrook 2006] SABETZADEH, Mehrdad ; EASTERBROOK, Steve: View merging in the presence of incompleteness and inconsistency. In: *Requirements Engineering* 11 (2006), p. 174–193

[Schaathun and Rutle 2018] Schaathun, Hans G. ; Rutle, Adrian: Model-driven engineering in RDF-a way to version control. In: *Norsk IKT-konferanse for forskning og utdanning*, 2018

[Schipper et al 2009] Schipper, A. ; Fuhrmann, H. ; Hanxleden, R. von: Visual comparison of graphical models. In: *International Conference on Engineering of Complex Computer Systems*, IEEE, 2009, p. 335–340

[Schmidt 2006] Schmidt, D. C.: Guest editor's introduction: Model-driven engineering. In: *Computer* 39 (2006), Nr. 2, p. 25–31

[Schröpfer et al 2019] Schröpfer, Johannes ; Schwägerl, Felix ; Westfechtel, Bernhard: Consistency control for model versions in evolving model-driven software product lines. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)* IEEE (event), 2019, p. 268–277

[Schultheiß et al 2021] Schultheiss, Alexander ; Bittner, Paul M. ; Grunske, Lars ; Thüm, Thomas ; Kehrer, Timo: 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)* IEEE (event), 2021, p. 1–12

[Schwägerl et al 2015] Schwägerl, F. ; Uhrig, S. ; Westfechtel, B.: A graph-based algorithm for three-way merging of ordered collections in EMF models. In: *Science of Computer Programming* 113 (2015), p. 51–81

[Sharbaf and Zamani 2017] Sharbaf, Mohammadreza ; Zamani, Bahman: A UML profile for modeling the conflicts in model merging. In: *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)* IEEE (event), 2017, p. 0197–0202

[Sharbaf and Zamani 2020] Sharbaf, Mohammadreza ; Zamani, Bahman: Configurable three-way model merging. In: *Software: Practice and Experience* 50 (2020), Nr. 8, p. 1565–1599

[Sharbaf et al 2015] Sharbaf, Mohammadreza ; Zamani, Bahman ; Ladani, Behrouz T.: Towards automatic generation of formal specifications for UML consistency verification. In: *2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI)* IEEE (event), 2015, p. 860–865

[Sharbaf et al 2020] Sharbaf, Mohammadreza ; Zamani, Bahman ; Sunyé, Gerson: A Formalism for Specifying Model Merging Conflicts. In: *Proceedings of the 12th System Analysis and Modelling Conference*, 2020, p. 1–10

[Sharbaf et al 2022a] Sharbaf, Mohammadreza ; Zamani, Bahman ; Sunyé, Gerson: Automatic resolution of model merging conflicts using quality-based reinforcement learning. In: *Journal of Computer Languages* 71 (2022), p. 101123

[Sharbaf et al 2022b] Sharbaf, Mohammadreza ; Zamani, Bahman ; Sunyé, Gerson: Conflict management techniques for model merging: a systematic mapping review. In: *Software & Systems Modeling* 22 (2022), October, Nr. 3, p. 1031–1079

[Shen et al 2021] Shen, Bo ; Zhang, Wei ; Yu, Ailun ; Shi, Yifan ; Zhao, Haiyan ; Jin, Zhi: SoManyConflicts: Resolve Many Merge Conflicts Interactively and Systematically. In: *Automated Software Engineering*, IEEE, 2021, p. 1291–1295

[Shen et al 2019] Shen, Bo ; Zhang, Wei ; Zhao, Haiyan ; Liang, Guangtai ; Jin, Zhi ; Wang, Qianxiang: IntelliMerge: a refactoring-aware software merging technique. In: *Proceedings of the ACM on Programming Languages* 3 (2019), Nr. OOPSLA, p. 1–28

[Simulink 2023] Simulink: `https://www.mathworks.com/help/simulink/ug/about-simulink-model-comparison.html`. 2023. – (last accessed in March 2023)

[Sirius 2023a] Sirius: `https://eclipse.dev/sirius`. 2023. – (accessed august 2023)

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

[Stephan and Cordy 2013] Stephan, M. ; Cordy, J. R.: A survey of model comparison approaches and applications. In: *MODELSWARD*, SciTePress, 2013, p. 265–277

[Störrle 2010] Störrle, Harald: Towards Clone Detection in UML Domain Models. New York, NY, USA : Association for Computing Machinery, 2010 (ECSA '10), p. 285–293. – ISBN 9781450301794

[Störrle 2017] Störrle, Harald: Cost-Effective Evolution of Research Prototypes into End-User Tools. 134 (2017), Nr. C, p. 47–60. – ISSN 0167-6423

[Strüber et al 2017] Strüber, Daniel ; Born, Kristopher ; Gill, Kanwal D. ; Groner, Raffaela ; Kehrer, Timo ; Ohrndorf, Manuel ; Tichy, Matthias: Henshin: A usability-focused framework for EMF model transformation development. In: *International Conference on Graph Transformation*, Springer, 2017, p. 196–208

[SVN last accessed 2023]

[Syriani and Vangheluwe 2013] Syriani, E. ; Vangheluwe, H.: A modular timed graph transformation language for simulation-based design. In: *Software & System Modeling* 12 (2013), Nr. 2, p. 387–414

[Syriani et al 2013] Syriani, E. ; Vangheluwe, H. ; Mannadiar, R. ; Hansen, C. ; Van Mierlo, S. ; Ergin, H.: AToMPM: A web-based modeling environment. In: *Companion proceedings* Volume 1115, CEUR-WS.org, 2013, p. 21–25

[SystemWeaver 2023] SystemWeaver: `https://support.systemweaver.se/support/solutions/articles/31000156469-versioning-in-systemweaver`. 2023. – (last accessed in March 2023)

[Taentzer et al 2010] Taentzer, Gabriele ; Ermel, Claudia ; Langer, Philip ; Wimmer, Manuel: Conflict detection for model versioning based on graph modifications. In: *Graph Transformations: 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27–October 2, 2010. Proceedings 5* Springer (event), 2010, p. 171–186

[Taentzer et al 2014] Taentzer, Gabriele ; Ermel, Claudia ; Langer, Philip ; Wimmer, Manuel: A fundamental approach to model versioning based on graph modifications: from theory to implementation. In: *Software & Systems Modeling* 13 (2014), Nr. 1, p. 239–272

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

[Tarjan 1972] Tarjan, Robert: Depth-first search and linear graph algorithms. In: *SIAM journal on computing* 1 (1972), Nr. 2, p. 146–160

[Toyoshima et al 2015] Toyoshima, Ichiro ; Yamaguchi, Shingo ; Zhang, Jia: A Refactoring Algorithm of Workflows Based on Petri Nets. In: *International Congress on Advanced Applied Informatics*, IEEE, 2015, p. 79–84

[Tröls et al 2019] Tröls, Michael A. ; Mashkoor, Atif ; Egyed, Alexander: Live and global consistency checking in a collaborative engineering environment. In: *Proceedings of* *the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, p. 1776–1785

[Tröls et al 2021] TRÖLS, Michael A. ; MASHKOOR, Atif ; EGYED, Alexander: Hierarchical distribution of consistency-relevant changes in a collaborative engineering environment. In: *2021 IEEE/ACM Joint 15th International Conference on Software and System Processes (ICSSP) and 16th ACM/IEEE International Conference on Global Software Engineering (ICGSE)* IEEE (event), 2021, p. 83–93

[Tsantalis et al 2020] TSANTALIS, Nikolaos ; KETKAR, Ameya ; DIG, Danny: Refactoring-Miner 2.0. In: *Transactions on Software Engineering* 48 (2020), Nr. 3, p. 930–950

[de la Vega and Kolovos 2022] VEGA, Alfonso de la ; KOLOVOS, Dimitris: An efficient line-based approach for resolving merge conflicts in XMI-based models. In: *Software and Systems Modeling* (2022), p. 1–27

[Vermolen et al 2012] VERMOLEN, S. D. ; WACHSMUTH, G. ; VISSER, E.: Reconstructing complex metamodel evolution. In: *SLE* Volume 6940, Springer, 2012, p. 201–221

[Völter and Kelly 2013] VÖLTER, Markus ; KELLY, Steven: *Model-Driven Development: A Practical Approach*. Lulu.com, 2013

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

[Westfechtel 2014] WESTFECHTEL, Bernhard: Merging of EMF models: Formal foundations. In: *Software & Systems Modeling* 13 (2014), p. 757–788

[Wieland et al 2013] WIELAND, Konrad ; LANGER, Philip ; SEIDL, Martina ; WIMMER, Manuel ; KAPPEL, Gerti: Turning conflicts into collaboration. In: *Computer Supported Cooperative Work (CSCW)* 22 (2013), Nr. 2, p. 181–240

[Xing and Stroulia 2005] XING, Zhenchang ; STROULIA, Eleni: UMLDiff: an algorithm for object-oriented design differencing. In: *Automated Software Engineering*, ACM, 2005, p. 54–65

[xtend last accessed 2023] XTEND: `https://www.eclipse.org/xtend/index.html`. last accessed 2023. – (accessed August 2023)

[Xtext last accessed 2023] XTEXT: `https://eclipse.dev/Xtext`. last accessed 2023. – (accessed august 2023)

[Zadahmad et al 2019] ZADAHMAD, Manouchehr ; SYRIANI, Eugene ; ALAM, Omar ; GUERRA, Esther ; LARA, Juan de: Domain-Specific Model Differencing in Visual Concrete

Syntax. In: *Software Language Engineering*. Athens : ACM, oct 2019, p. 100–112

[Zadahmad et al 2022] Zadahmad, Manouchehr ; Syriani, Eugene ; Alam, Omar ; Guerra, Esther ; Lara, Juan de: DSMCompare: Domain-Specific Model Differencing for Graphical Domain-Specific Languages. In: *Software & Systems Modeling* 21 (2022), p. 2067–2096

[Zerrouk et al 2018] Zerrouk, Manar ; Anwar, Adil ; Benelallam, Imade: Managing model conflicts in collaborative modeling using constraint programming. In: *2018 IEEE 5th International Congress on Information Science and Technology (CiSt)* IEEE (event), 2018, p. 117–123