A Formalism for Specifying Model Merging Conflicts

Mohammadreza Sharbaf, Bahman Zamani, Gerson Sunyé

▶ **To cite this version:**

Mohammadreza Sharbaf, Bahman Zamani, Gerson Sunyé. A Formalism for Specifying Model Merging Conflicts. System Analysis and Modelling (SAM) conference, Oct 2020, Virtual Event, Canada. 10.1145/3419804.3421447 . hal-02930770

HAL Id: hal-02930770
https://hal.science/hal-02930770

Submitted on 4 Sep 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Formalism for Specifying Model Merging Conflicts

Mohammadreza Sharbaf *
MDSE Group, University of Isfahan
Isfahan, Iran
m.sharbaf@eng.ui.ac.ir

Bahman Zamani
MDSE Group, University of Isfahan
Isfahan, Iran
zamani@eng.ui.ac.ir

Gerson Sunyé
LS2N, University of Nantes
Nantes, France
gerson.sunye@univ-nantes.fr

ABSTRACT

Verifying the consistency of model merging is an important step towards the support for team collaboration in software modeling and evolution. Since merging conflicts are inevitable, this has triggered intensive research on conflict management in different domains. Despite these efforts, techniques for high-level conflict representation have hardly been investigated yet. In this paper, we propose an approach to specify model merging conflicts. This approach includes the Conflict Pattern Language (CPL), a formalism for specifying conflicts in different modeling languages. CPL is based on the OCL grammar and is tooled by an editor and a parser. CPL facilitates the slow and error-prone task of specifying model merging conflicts and can be used to specify conflicts in any EMF-based model. We evaluated our approach with a case study, including five different conflict cases. The results are promising about how CPL can be used for specifying syntactic and semantic conflicts.

CCS CONCEPTS

• **Software and its engineering** → **Specification languages**; **Formal language definitions**.

KEYWORDS

Model Merging Conflict, Conflict Specification Formalism, Conflict Representation, Collaborative Modeling, Model Driven Engineering

ACM Reference Format:
Mohammadreza Sharbaf, Bahman Zamani, and Gerson Sunyé. 2020. A Formalism for Specifying Model Merging Conflicts. In *12th System Analysis and Modelling Conference (SAM '20), October 19–20, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3419804.3421447

1 INTRODUCTION

Software development includes several activities, analysis, design, and implementation, to name a few. These activities are mainly complex and require collaboration between different experts, coming from a wide variety of fields [32]. Modeling, which is mainly used in the analysis and design phases of software development is no exception. When the software is complex, the size and complexity of models increases dramatically. This enforces that many heterogeneous partners participate in large teams and work collaboratively together [22]. Some of the participants may work concurrently and independently on the same model from different geographical sites. Each of the participants focuses on specific aspects of the system and locally modifies only a particular part of the model. When participants deliver locally modified models, those need to be integrated into a common relevant model for continuing the software evolution. The problem is that some concurrent updates may be incompatible and contribute to conflicts during the integration process. Therefore, it is critical to focus on conflict management to maintain the model consistent [11].

To support conflict management in model merging, several approaches have been proposed so far. Most of them provide conflict detection techniques to discover conflicts which may occur due to concurrent changes. Some of them focus on reconciling conflicts with context-free techniques based on a three-way model merging strategy [11]. Furthermore, some approaches try to prevent conflict occurrence by using mechanisms to inform the user about conflicting situations for concurrent actions [17]. Despite these efforts, techniques for high-level conflict representation have been rarely studied yet. The fact is that, for performing conflict management activities, we have to know which situation is considered as a conflict. In merging models, we face different kinds of conflicts. Each conflict refers to a particular context that depends on the syntax and semantics of the modeling language [1]. Potential conflicts are mainly domain-specific; hence, they can be hard to understand and be recognized solely by language engineers. However, in the collaborative evolution of models, in merging new versions, it is necessary to automatically check for potential conflicts, which is not possible without a precise or formal specification of conflicts. To this end, an approach to specify merging conflicts for different modeling languages is needed, which is still remained as an issue.

In this paper, we present a formalism for specifying model merging conflicts to solve the issue previously mentioned. Our proposal is based on the concept of patterns as approved solutions to recurring specification and design problems. We define each conflict as a pattern that describes the situation of model elements at the time of conflict. To propose the conflict representation formalism, we provide an extension to the Graphical Extension of BNF (GEBNF) [3]. Based on this formalism, we designed a language called Conflict Pattern Language (CPL), which applies to any EMF-based model. To facilitate the slow and error-prone task of writing and syntactically validating the specification of model merging conflict patterns, we also extended the Object Constraint Language (OCL) grammar to provide a grammar parser and a syntax-aware editor.

The proposed conflict representation approach is illustrated by the "Pull-Up Method" refactoring conflict [19] for the UML Class diagram; however, it is applicable to specify conflict patterns for models conforming to arbitrary metamodels. To evaluate the applicability of our proposed formalism and language, we ran a case study to answer the following research questions:

*Also with LS2N, University of Nantes, Nantes, France.

SAM '20, October 19–20, 2020, Virtual Event, Canada
© 2020 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *12th System Analysis and Modelling Conference (SAM '20), October 19–20, 2020, Virtual Event, Canada,* https://doi.org/10.1145/3419804.3421447.

RQ1) Is it possible to specify any model merging conflict as a pattern?
RQ2) Is the proposed formalism able to represent model merging conflicts?
RQ3) Is our formalism able to support the representation of conflicts for different modeling languages?

The rest of this paper is organized as follows. Section 2 briefly provides background information on collaborative modeling as well as model merging conflicts and introduces our running example. Section 3 provides the idea of using patterns for representing model merging conflicts and presents the conflict specification formalism. The combination of conflict pattern specification with OCL to implement the Conflict Pattern Language (CPL) parser and editor is reported in Section 4. Section 5 presents the case study which is amended by a discussion on each research question. Section 6 reports related work. Finally, Section 7 concludes the paper and highlights areas for future work.

2 BACKGROUND

2.1 Collaborative Modeling

In software development, different stakeholders collaborate to build complex systems. This collaboration results in the specification, architecture, and design of the system, each of which can be considered as a model of the system. Stakeholders use specific modeling languages to build these models. The joint creation of models is considered as collaborative modeling [24].

In collaborative modeling, change propagation occurs in both, offline or online scenarios [14]. Offline collaboration is based on asynchronous interactions. In this scenario, users check out models from a version control system (VCS) and commit local changes to the repository.

In the online scenario, users work synchronously and may simultaneously edit a model. Changes are immediately propagated to all users. While changes are propagated in both offline and online scenarios, appropriate mechanisms for conflict management are required in destinations [17]. This includes support for (a) *conflict detection* in which potential conflicts are discovered, (b) *conflict awareness* that warns users of potential conflicts, and (c) *conflict resolution* by which detected conflicts are fixed.

2.2 Model Merging Conflicts

In the model merging process, conflicts may arise when concurrent incompatible modifications are applied to the same element. In such a situation, either the modifications cannot be integrated to produce a unique model, or the integration would result in an inconsistent merged version of the model. The main reason for conflicts is the existence of contradicting changes, which do not commute [5]. However, different types of conflicts in the merge of models can be defined by considering syntax and semantics of models [18].

Syntactic conflicts are those which take the modeling language syntax into account. However, syntactic conflicts may not necessarily produce a language syntax violation. For instance, contradicting updates to the name of the same element constitutes a syntactic conflict. This type of conflict may be detected by checking the structure of models and comparing the similarity of elements.

Semantic conflicts go beyond the generic conflicts and need to consider the system behavior and the modeling language semantics. Indeed, the integration of concurrent changes may result in a syntactically correct merged version, yet semantically invalid [1]. Semantic conflicts are divided into three categories: Static semantics, behavioral semantics, and semantic equivalence [2].

In this context, *static semantic* conflicts refer to issues and side effects detected at compile-time, such as violation of hierarchy constraints and incompatible types, which remain hidden in the merged version without considering the model semantics. In most modeling languages, e. g. UML, these conflicts violate their well-formedness rules and are perceived as syntax conflicts. *Behavioral semantic* conflicts denote different or unexpected behavior in the merged model, which relies on the runtime semantics. It considers the execution behavior of modeled systems based on the *data* and *control flow*. *Semantic equivalence* conflicts denote contradicting conditions in which the modeler can express the same meaning in different ways using equivalent concepts or equivalent constructs. In those conditions, syntactically, both modifications can be applied because of their different appearance, while only one of them should be applied to perform a valid merged model.

2.3 Running Example: the Pull Up Method Refactoring Conflict

One conflict that may occur in a UML Class Diagram is when the designer applies the *Pull Up Method* refactoring operation [31]. We use this refactoring to illustrate the idea of using patterns to specify and formalize model merging conflict representation.

Figure 1 shows an instance of the Pull-Up Method conflict for different versions of an Order Management System [28]. The Pull-Up Method conflict occurs when two modelers apply two different actions on methods that are common among all subclasses of a superclass in the original version. On the left hand, the first modeler moves the *authorized* method from the subclasses to the superclass. On the right hand, the second modeler creates a new class *Cash*, which is inherited from the class *Payment* as the superclass. The newly added subclass does not have the common method (*authorized*). In this case, at the merged version, the *authorized* method, as a common feature, is added only to the superclass (*Payment*). Also, the *Cash* class is added as the new subclass of class *Payment* in the merged version. Consequently, the merged version is semantically incorrect because the new subclass (*Cash*) inherits the *authorized* method, while this is not what the second modeler meant to do.

3 CONFLICT SPECIFICATION FORMALISM

3.1 Specifying Conflict by Pattern

In the parallel modification of a model, users apply a series of changes that turn the model into different versions. After various modifications, these versions may have non-similar parts, which at the merge process may lead to the failure of integration or result in an invalid merged model. More precisely, the structure and situation of elements in different versions lead to a model merging conflict. Hence, a model merging conflict can be identified by describing the situation of the parts of each version that are involved in the conflict.

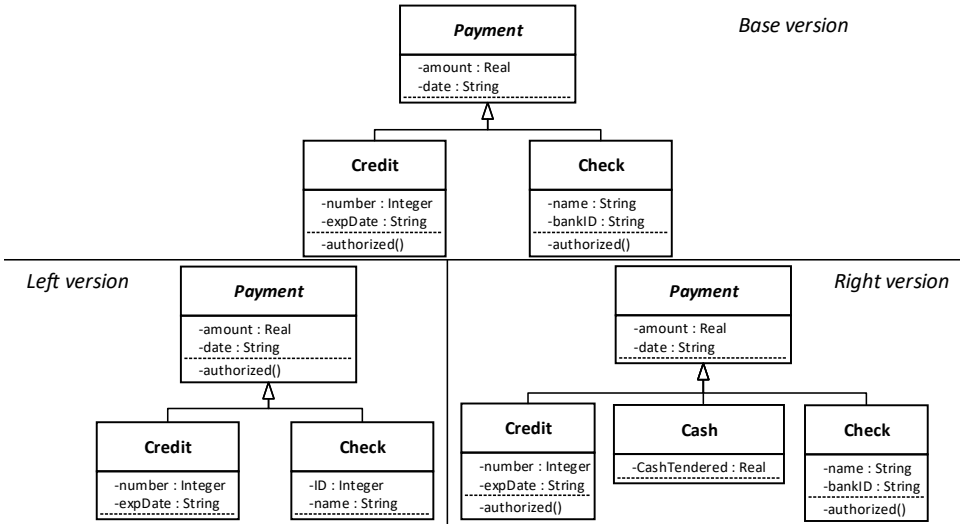

Figure 1: An Example of the Pull Up Method Conflict in Class Diagrams (adapted from [28])

Patterns express the structure and characteristics of repeated or regular situations. Therefore, in conflict management, patterns can describe detectable conflicts, expressing the structures, elements, and the relationships between the elements of the different versions that led to a conflict. To this end, we propose a template for specifying a model merging conflicts as patterns. It should be noted that the proposed pattern is only an approach for representing the model merging conflict; it does not provide a solution to resolve that conflict.

```
1  conflict <name>{
2     domain <domainName>:<domainPath>
3           (, <domainName>:<domainPath>)*
4     (inmodel <modelName>{
5          component (, component)*
6          (, statement)*)?
7     })+
8     (where {
9          condition (, (or|and) condition)*
10    })?
11    (description (:<expression>))?
12 }
```

Listing 1: Template of a Conflict Pattern

Listing 1 illustrates the template of a conflict pattern, where a named *conflict* block contains several named *domain* and *inmodel* statement blocks. Each *domain* has a *name* and a *path*. The domain *name* allows the identification of all elements belonging to that domain, while the domain *path* specifies a URI that contains a domain modeling language. Each *inmodel* block describes components and statements that identify sets of model elements with specific properties. Each conflict pattern description also includes an optional *where* condition, which describes the relationship among elements in the specified *inmodel* blocks. Finally, the optional *description* block defines a diagnostic message describing the situation that satisfies this pattern. In its simplest form, a conflict pattern consists of two *inmodel* statement blocks, which include the description of elements in two different versions.

In the following section, we formalize the proposed template for specifying conflict patterns to investigate in more detail the *component*, *statement*, *condition*, and *expression* blocks, which are introduced in Listing 1.

3.2 Conflict Pattern Formalism Using GEBNF

In this section, we specify the conflict pattern formalism using a set of rules in the meta-notation GEBNF [34], inspired from Rouhi and Zamani specification scheme [25].

Hereafter, these rules are indicated from Rule 1 to Rule 26. In these rules, each non-terminal symbol is reachable from the *ConflictPattern* root, and there exists one and only one description rule for each non-terminal symbol. These facts show that the proposed notation satisfies the *Reachability* and *Completeness* conditions, respectively. Hence, the conflict pattern formalism is a well-formed syntax [33].

Rule 1 defines that a model merging conflict pattern, as mentioned in the template of conflict pattern (Listing 1), includes (i) a *name*, (ii) the *domains* of involved models, (iii) the different *fragments* of models that raise the conflict, (iv) the *relations* between model elements in the different fragments, and finally, (v) the *description* that should be displayed to the user. According to Rule 1, each pattern starts with the *'conflict'* keyword, followed by the name of conflict pattern and then the parts that specify the body of the pattern and are surrounded by braces (*'{'* and *'}'*).

$$\begin{aligned} \textit{ConflictPattern} \quad ::= \quad & \text{`}\textit{conflict}\text{'}, \ \textbf{patternName} : \textit{String}, \text{`}\{\text{'}, \\ & \quad \textbf{modelDomain} : \textit{domainExpression}^{+}, \\ & \quad \textbf{modelFragment} : \textit{fragDeclaration}^{+}, \\ & \quad \textbf{modelRelations} : [\textit{relStatement}], \\ & \quad \textbf{conflictDescription} : [\textit{dscExpression}], \\ & \text{`}\}\text{'} \end{aligned} \tag{1}$$

In the body of a pattern, the *domainExpression* field (Rule 2) defines the modeling language that contextualizes the conflict pattern.

Each domain expression consists of a *'domain'* keyword, a string name (Rule 3), and an *address* (Rule 4), which is a URI to access the definition of the specified modeling language.

$$domainExpression ::= \text{`}domain\text{'}, domainName, \text{` : `}, address \quad (2)$$

$$domainName ::= name : String \quad (3)$$

$$address ::= \text{` " `}, uri : String, [\text{`/'}, id : String]^*, \text{` " `} \quad (4)$$

The *fragDeclaration* field (Rule 5) defines the constituent sub-models. Each fragment declaration consists of the '*inmodel*' keyword, the desired *model*, and a number of *declarations* and optional *conditions* to specify a set of model elements.

$$fragDeclaration ::= \text{`}inmodel\text{'}, \textbf{model} : modelContext, \textbf{declaration} : modelElement^+, \textbf{condition} : condFormula^* \quad (5)$$

In each *fragDeclaration*, the *modelContext* (Rule 6) includes a string *modelName* and a reference *domainName*, which refers to a domain that is specified by a *domianExpression* (Rule 2). The *modelElement* (Rule 7) consists of a string *name* and a collection of model elements that can be defined as a *inclusionSet* (Rule 8), e. g., $\{source, target\} \subseteq Node$ or a *membershipSet* (Rule 9), e. g., $c \in Class$. The *Member* (Rule 10) is a single named identifier or an ordered list of them. The classifier and variable identifiers, or attribute and function names, are instances of the *Member* rule. Rule 11 specifies the different types of model elements, which can be *EDataType*, *EClass*, or power-set type, to support all Ecore. According to this rule, the user can also introduce other user-defined types by a named string. The *condition* field as the last part of *fragDeclaration* rule is defined by *condFormula* (Rule 12). The conditional formula is either a positive or a negative formula which can be defined as *simpleFormula* (Rule 13), *quantFormula* (Rule 18), or a conjunction/disjunction composition of them.

$$modelContext ::= modelName : String, \text{` : `}, \underline{domainName} \quad (6)$$

$$modelElement ::= name : String, \text{` = `}, inclusionSet \mid membershipSet \quad (7)$$

$$inclusionSet ::= \text{`\{'}, Member, [\text{`, '}, Member]^*, \text{`\}'}, \text{` } \subseteq \text{ `}, Type \quad (8)$$

$$membershipSet ::= Member, \text{` } \in \text{ `}, Type \quad (9)$$

$$Member ::= name : String, [\text{`('}, [id : String, [\text{`('}, \text{`)'}], [\text{`, '}]]^*, \text{`)'}] \quad (10)$$

$$Type ::= EDataType \mid EClass \mid \text{` } \mathbb{P} \text{ `}, Type \mid id : String \quad (11)$$

$$condFormula ::= [\text{`} \neg \text{ `}], simpleFormula \mid quantFormula \mid \text{`('}, [[\text{` } \vee \text{ `}], condFormula]^* \mid [[\text{` } \wedge \text{ `}], condFormula]^*, \text{`)'} \quad (12)$$

A *simpleFormula* (Rule 13) consists of two *Operands* (Rule 14) interleaved by a comparison operator, introduced in the *comparisonOp* (Rule 15). Each *Operand* is either a single user-defined identifier that is specified by a string name or a collection of elements that are expressed by a *inclusionSet* (Rule 8), a *membershipSet* (Rule 9), or a *Function* (Rule 16). The *Function* consists of a return *Type* (Rule 13), a string name, and an ordered list of input parameters that are defined by the *Param* (Rule 17). Each parameter consists of a user-defined identifier, whose type is defined based on the *Type* rule after the ' : ' notation.

$$simpleFormula ::= Operand, comparisonOp, Operand \quad (13)$$

$$Operand ::= name : String \mid membershipSet \mid inclusionSet \mid Function \quad (14)$$

$$comparisonOp ::= \text{` < `} \mid \text{` } \leq \text{ `} \mid \text{` > `} \mid \text{` } \geq \text{ `} \mid \text{` = `} \mid \text{` <> `} \mid \text{` } \in \text{ `} \mid \text{` } \notin \text{ `} \quad (15)$$

$$Function ::= Type, name : String, \text{`('}, (Param, [\text{`, '}])^*, \text{`)'} \quad (16)$$

$$Param ::= id : String, \text{` : `}, Type \quad (17)$$

The *quantFormula* (Rule 18) introduces the quantifier formula, which consists of a *Quantifier* symbol (' $\forall$ ' or ' $\exists$ '), a list of variables with a specific type based on the *varType* (Rule 20), an optional such that *condFormula* preceded by a ' | ', an optional consequent symbol (' $\Rightarrow$ ' or ' $\Leftrightarrow$ '), and a *condFormula*, which provides the capability of introducing quantifier formula by the recursive form.

$$quantFormula ::= [Quantifier, varType]^+, [\text{` | `}, condFormula], [ConseqCon], condFormula \quad (18)$$

$$Quantifier ::= \text{` } \forall \text{ `} \mid \text{` } \exists \text{ `} \quad (19)$$

$$varType ::= id1 : String, [\text{`, '}, id2 : String]^*, \text{` : `}, Type \quad (20)$$

$$ConseqCon ::= \text{` } \Rightarrow \text{ `} \mid \text{` } \Leftrightarrow \text{ `} \quad (21)$$

In the *ConflictPattern* rule (Rule 1), the *relStatement* field expresses the relationships between the collected elements in different *modelFragments*. The *relStatement* (Rule 22) is defined as either a negative or a positive statement preceded by the '*where*' keyword. The statement is defined as a *simpleRelation* (Rule 23), a *quantRelation* (Rule 25), or a conjunction/disjunction composition of them. The *simpleRelation* consists of two *modelElements* that are compared by an operator based on the *relCompOp* (Rule 24). Each *modelElement* refers to a defined element in a *modelFragment* (Rule 5). The reference to a *modelElement* consists of a *modelContext* name and a number of *modelElements'* names, chained by dots. The *relCompOp* (Rule 24) consists of all comparison operators of *comparisonOp* (Rule 15) and two new operators, '*isEquivalent*' and '*isContradict*', which can be used to compare the semantics of two operands.

Finally, the *quantRelation* (Rule 25) is a list that consists of a *quantifier* symbol (' $\forall$ ' or ' $\exists$ '), a list of *varType* variables (Rule 20), an optional *relStatement* preceded by a ' | ', an optional consequent symbol (' $\Rightarrow$ ' or ' $\Leftrightarrow$ '), and a *relStatement*.

$$relStatement ::= \text{`}where\text{'}, [\text{`} \neg \text{ `}], simpleRelation \mid quantRelation \mid \text{`('}, [[\text{` } \vee \text{ `}], relStatement]^* \mid [[\text{` } \wedge \text{ `}], relStatement]^*, \text{`)'} \quad (22)$$

$$simpleRelation ::= \underline{modelContext}, [\text{`.'}, \underline{modelElement}]^+, relCompOp, \underline{modelContext}, [\text{`.'}, \underline{modelElement}]^+ \quad (23)$$

$$relCompOp ::= comparisonOp \mid \text{`}isEquivalent\text{'} \mid \text{`}isContradict\text{'} \quad (24)$$

$$quantRelation ::= [Quantifier, varType]^{+}, [\text{`}\mid\text{'}, relStatement], [ConseqCon], relStatement \quad (25)$$

At last, the *dscExpression* field (Rule 26) defines a diagnostic message, which includes the '*description*' keyword followed by the ' : ' notation and a textual expression.

$$dscExpression ::= \text{`}description\text{'}, \text{` : '}, expression : String \quad (26)$$

3.3 Formalizing the Pull-Up Method Conflict

To clarify the presented formalism for conflict pattern specification, let us refer to the illustrated running example. In this running example, we introduced a model merging conflict in the *domain* of UML class diagram, named the Pull-Up Method conflict. This conflict consists of three model *fragments*: *Base*, *Left*, and *Right*. Each fragment expresses a collection of involved classes in a specific inheritance relationship, including superclass and subclasses.

In the *Base* fragment, all subclasses must contain a common method. In the *Left* fragment, only the superclass has the common method, and in the *Right*, there is at least one subclass that does not contain the common method.

To raise a Pull-Up Method conflict, the common method, and superclass, which are specified in the *Base*, *Left*, and *Right* fragments, must be the same. The *modelRelations* part of the Pull-Up method conflict pattern is used to specify the similarity between superclass and common method of three fragments. Also, in the last part of the pattern, a textual *description* can be added to introduce the Pull-Up conflict pattern to the user.

In the following, we sketch the Pull-Up Method conflict pattern specification using the proposed GEBNF formalism. According to Rule 1, the specification consists of five parts that are presented separately. To show which rule is applied for each part, we added next to each predicate a comment containing the related rule numbers.

- **patternName:** `//Rule 1`
  conflict *Pull-Up Method*

- **modelDomain:** `//Rules 1-4`
  domain *UML : "http://www.eclipse.org/uml2/5.0.0/UML"*

- **modelFragment:** `//Rules 1,5`
  inmodel *Base : UML* `//Rule 6`
  $set1 = sc \in Class, set2 = op \in Operation$ `//Rules 7,9-11`
  $\forall c_1 : Class \mid (c_1 \in subclassOf(sc))$ `//Rules 11-21`
  $\Rightarrow \exists op_1 : Operation \mid op == op_1 \wedge c_1 \in (ownerOf(op_1))$
  inmodel *Left : UML* `//Rule 6`
  $set1 = sc \in Class, set2 = op \in Operation$ `//Rules 7,9-11`
  $\forall c_1 : Class \mid (c_1 \in subclassOf(sc))$ `//Rules 11-20`
  $\wedge (ownerOf(op) == sc)$
  inmodel *Right : UML* `//Rule 6`
  $set1 = sc \in Class, set2 = op \in Operation$ `//Rules 7,9-11`
  $\exists c_1 : Class \mid (c_1 \in subclassOf(sc) \wedge c_1 \notin (ownerOf(op)))$ `//Rules 11-20`

- **modelRelations:** `//Rules 1,22`
  where $(Left.set1.sc = Base.set1.sc$ `//Rules 15,22-24`
  $\wedge\ Base.set1.sc = Right.set1.sc$
  $\wedge\ Left.set2.op = Base.set2.op$
  $\wedge\ Base.set2.op = Right.set2.op)$

- **conflictDescription:** `//Rule 1,26`
  description: *"Due to concurrent modifications, a class inherits some methods which were not inherited from the original model!"*

4 TOOL SUPPORT

4.1 Enriching Conflict Specification with OCL

In section 3.2, we introduced a formalism for specifying model merging conflicts. Based on this formalism, we design and implement a language named the Conflict Pattern Language (CPL). The implementation provides language tool support, including a syntax-aware editor and a parser, which can be used by modelers or by any person in charge of model merging.

To enable conflict specification on models conforming to arbitrary metamodels, we need to address the vital issue of expressing model elements in the *modelFragment* part of conflict pattern formalism. A possible solution to declare the condition of specific elements is the Object Constraint Language, OCL.

OCL is one of the well-known languages in the modeling community, which is frequently used for validating and querying UML and EMF models [13, 20]. OCL provides a convenient set of logical operators and predicate operations for single elements and collections, which can facilitate the formal specification of model properties in a comprehensible way. Additionally, OCL allows the specification of invariants for any modeling element. Since the latest version of its specification [20], OCL is based on the common core of MOF and Ecore: it applies indiscriminately to UML models or to any model conforming to an arbitrary EMF metamodel.

We propose to use OCL Queries in the *modelFragment* part of conflict pattern, to express the specific fragments of models. To this purpose, we should reuse most of the *Essential OCL* notations in CPL. Essential OCL is an extension of OCL, which is closer to support EMF-based modeling technologies [23]. However, reusing the Essential OCL is not enough, since we need to add some new operations for describing semantic relationships in the *modelRelations* part. Therefore, we extend OCL operations by adding *isEquivalent* and *isContradict* operations. The *isEquivalent* operation expresses different modifications with the same meaning, whereas the *isContradict* operation describes elements that are changed in parallel by semantically or linguistically different modifications.

4.2 The Conflict Pattern Language

To facilitate the slow and error-prone process of writing and syntactically validating the specification of conflict patterns, we have developed the Conflict Pattern Language (CPL) as a set of plug-ins for Eclipse. CPL is available from GitHub[1] under the EPL 2.0 license. CPL is implemented using Xtext [4], a language workbench that provides automatic generation of parsers and syntax-aware editors from grammar specifications.

[1] https://github.com/MSharbaf/CPL2020

CPL.xtext

```
grammar org.xtext.example.cpl.CPL
        with org.eclipse.ocl.xtext.completeocl.CompleteOCL
import "http://www.eclipse.org/ocl/2015/BaseCS" as base
import "http://www.eclipse.org/ocl/2015/EssentialOCLCS" as eOCL
import "http://www.eclipse.org/ocl/2015/CompleteOCLCS" as cOCL
generate cPL "http://www.xtext.org/example/cpl/CPL"

TopLevelCP:
    'conflict' name=Identifier '{'
        (ownedDomains+=DomainCP)+
        (ownedFragment+=DeclarationCP)+
        (ownedRelation+=StatementCP)?
        (ownedDescription+=DescriptionCP)?
    '}' ;

DomainCP returns base::ImportCS:
    'domain' (name=Identifier ':')?
        ownedPathName=URIPathNameCS (isAll?='::*')? ;

DeclarationCP:
    'inmodel' name=Identifier '{'
        ownedContexts+=ContextDeclCS*
    '}' ;

StatementCP:
    'where' '{'
        (('or'|'and')? ownedConditions+=ConditionCS)+
    '}' ;

DescriptionCP:
    'description' '{' ownedExpression=STRING '}' ;
```

Figure 2: An Excerpt of the CPL Xtext Grammar

One of the main advantages of Xtext is that the Eclipse Foundation already provides an OCL Xtext grammar. Since CPL is based on OCL, we first reused this grammar of OCL by a way of the *grammar mixin* mechanism. Then, we redefined the necessary parts of *Basic*, *EssentialOCL*, and *CompleteOCL* to enrich *modelFragment* part of CPL for performing queries against the UML and EMF models. Finally, we completed the CPL grammar and its syntax validator according to the specific rules and keywords in the conflict pattern formalism. CPL is able to suggest appropriate variables and operations on writing conflict specifications, which helps language engineers write the correct conflict specification faster. Figure 2 shows a partial Xtext grammar for CPL, which presents the *ConflictPattern* rule (Rule 1) of the CPL formalism.

Figure 3 illustrates the use of the developed editor to specify the Pull-Up Method conflict in CPL. Line 1 specifies the name of the conflict pattern and line 2 defines the domain metamodel, which is loaded under the local name *UML*. Then, lines 3-9 express a set of queries that will be applied to collect a set of elements in the *Base* version. As such, lines 10-14 and 15-21 express queries for identifying elements in *Left* and *Right* versions, respectively. Lines 22-25, specify the relationships between expressed elements that need to be satisfied to occur conflict. Finally, lines 26-28 provide the conflict description that explains the reason of conflict occurrence.

5 CASE STUDY

To evaluate our conflict specification formalism, we run a case study with a range of model merging conflicts. Our evaluation aims at investigating the applicability of our formalism and illustrates the conflict pattern expressiveness by examples. To this end, we chose a set of model merging conflicts that correspond to the different types of conflicts, including *syntactic*, *static semantics*, *behavioral semantics*, and *semantically equivalence*, presented in Section 2.2.

In the following, we introduce the following merge conflicts: *dangling reference*, *polyforest cycle*, *equivalent associations*, *data flow inconsistency*, and *control flow loop*. After each conflict description, we present its representation in CPL. Finally, we discuss the results and the answers to the research questions.

5.1 Conflict Cases

Dangling Reference. A syntactic conflict in the UML class diagram is the dangling reference conflict [2]. As illustrated in Figure 4, a dangling reference may arise when one user adds an association between the *Person* and *Car*, whereas the other user deletes the *Car* class. The representation of dangling reference conflict by CPL is illustrated in Listing 2.

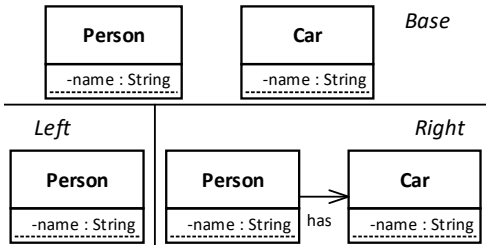

Figure 4: Dangling Reference: a Syntactic Conflict

```
1  conflict DanglingReference{
2      domain UML: 'http://www.eclipse.org/uml2/5.0.0/UML#/'
3      inmodel Base{
4       context UML::Class
5        def: bSet:Set(Class)=Class.allInstances()→select(c1|
6                    Class.allInstances()→exists(c2:Class|c1<>c2))
7      }
8      inmodel Left{
9       context UML::Class
10       def:lSet:Set(Class)=Class.allInstances()→select(c1|c1<>null)
11     }
12     inmodel Right{
13      context UML::Class
14       def: rSet:Set(Class)=Class.allInstances()→select(c1| Class.
15          allInstances()→exists(c2| c1<>c2 and c1.getAssociations()
16                   →intersection(c2.getAssociations())→notEmpty()))
17     }
18     where{
19       Base.bSet.c1=Right.rSet.c1 and Base.bSet.c1=Left.lSet.c1 and
20       Base.bSet.c2=Right.rSet.c2 and Base.bSet.c2 NotIN Left.lSet
21     }
22     description {
23     "One added an association to a class while other removed class."
24     }
25 }
```

Listing 2: CPL Specification of Dangling Reference Conflict

Polyforest Cycle. The occurrence of a cycle in a polyforest is an example of a static semantic conflict for an EMF-based model. A polyforest is a directed acyclic graph, known as a collection of directed trees with potentially multiple roots and no graph cycles [26]. Figure 5 illustrates an example of this conflict, in which models conform to a simple Graph metamodel. While there is no cycle in the base model version, if modelers add different edges that lead to a cycle in the merged version, the cyclic conflict would arise. Listing 3 illustrates the polyforest cycle conflict expressed in CPL.

PullUpMethod.cpl

```
conflict Pull_Up_Method{
    domain UML: 'http://www.eclipse.org/uml2/5.0.0/UML#/'
    inmodel Base{
        context UML::Class
        def: getOps:Collection(Operation)=self.generalization->asSequence()->first().general.getAllOperations()
        def: set1: Set(Class) = Class.allInstances()->select(sc:Class|sc.generalization->notEmpty()
            and sc.generalization->forAll(c1|c1.general.getAllOperations()->notEmpty() implies
            let set2 = sc.getOps->exists(op:Operation|c1.general.getAllOperations()->includes(op)) in set2<>null))
    }
    inmodel Left{
        context UML::Class
        def: set1: Set(Class) = Class.allInstances()->select(sc:Class|sc.generalization->notEmpty() and
            let set2 = sc.getAllOperations()->exists(op|op<>null) in set2.oclIsTypeOf(Operation))
    }
    inmodel Right{
        context UML::Class
        def: getAllSubOps:Collection(Operation)=self.generalization.general.getAllOperations()
        def: set1: Set(Class) = Class.allInstances()->select(sc:Class|sc.generalization->notEmpty() and
            let set2 = sc.generalization->exists(c1|sc.getAllSubOps->exists(op:Operation|
                sc.getAllSubOps->excludesAll(c1.general.getAllOperations())->includes(op))) in set2<>null)
    }
    where{
        Base.set1.sc = Left.set1.sc  and  Base.set1.sc = Right.set1.sc  and
        Base.set2.op = Left.set1.op  and  Base.set2.op = Right.set2.op
    }
    description{
      "Due to concurrent modifications, a class inherits some methods which were not inherited from the original model!"
    }
}
```

Figure 3: Specification of the Pull-Up Method Conflict in the CPL Editor

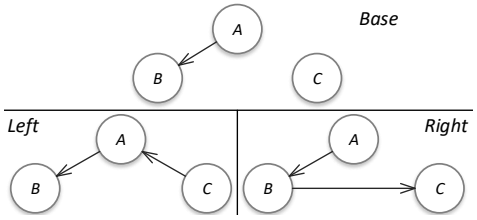

Figure 5: Polyforest Cycle: a Static Semantic Conflict

```
conflict PolyforestCycleConflict{
    domain Graph : '../TestProject/Graph.ecore#/'
    inmodel Left{
      context Graph::Node
       def: origin:Node = self
       def: lSet:Set(Node) = Node.allInstances()→select(
                    self.outgoing→closure(target.outgoing).target)
    }
    inmodel Right{
      context Graph::Node
       def: origin:Node = self
       def: rSet:Set(Node) = Node.allInstances()→select(
                    self.incoming→closure(source.incoming).source)
    }
    where{
      Left.origin=Right.origin and  Left.lSet.Node IN Right.rSet
    }
    description {
     "The merged concurrent modifications result in a cycle."
    }
}
```

Listing 3: CPL Specification of the Polyforest Cycle Conflict

Equivalent Associations. Figure 6 depicts a semantical equivalence conflict in UML class diagrams. In this example, adapted from [1], two users express the fact that each *Person* has *parents* in different ways. The first user adds two associations as the *mother* and *father* of a *Person* to create the left version. In his side, the second user adds an association to express that a *Person* has two *parents*. In this case, the merged version will contain three associations that express the same information. Hence, a semantical equivalence conflict should be reported. Listing 4 illustrates the CPL representation for this conflict.

```
conflict Equivalent_Associations{
    domain UML: 'http://www.eclipse.org/uml2/5.0.0/UML#/'
    inmodel Left{
     context UML::Association
      def: lSt:Set(Association) = Association.allInstances()
           →select(a:Association| Association.allInstances()
             →exists(a2| a.ownedEnd→at(1)=a2.ownedEnd→at(1) and
               a.ownedEnd→at(1)=a2.ownedEnd→at(2)) and a< >a2))
    }
    inmodel Right{
     context UML::Association
      def: rSt:Set(Association) = Association.allInstances()
                                  →select(a:Association|a< >null)
    }
    where{
      Left.lSt.a.ownedEnd = Right.rSt.a.ownedEnd and
      Left.lSt isEquivalent Right.rSt.a
    }
    description {
     "Some associations define the same facts in different ways."
    }
}
```

Listing 4: CPL Spec. of the Equivalent Associations Conflict

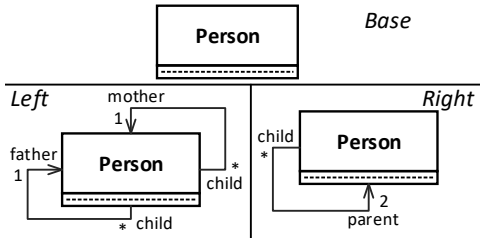

Figure 6: Equivalent Associations: a Semantical Equivalence Conflict

Data Flow Inconsistency. Figure 7 shows a behavioral semantic conflict for WSBPEL models, adapted from [1]. WSBPEL is a language for specifying business process behavior as web services [12]. In Figure 7, the WSBPEL model expresses the calculation of payment fees for a shopping system as the variable *sum*, which is equal to the summation of the *charge* and *tax* variables. In this example, the concurrent modification of *charge* and *tax* variables cause inconsistency in the value of variable *sum*, which leads to a behavioral semantic conflict due to inequality in the expected output for the variable *sum*. This conflict can be specified by expressing the condition and checking the data value of *sum* variable, which should be the same in the new versions. Listing 5 the CPL representation for this conflict.

```
conflict Data_Flow_Inconsistency{
    domain WSBPEL : '../TestProject/bpel.ecore#/'
    inmodel Base{
     context WSBPEL::Assign
      def: bS:Assign=Assign.allInstances()→select(a:Assign|
          a.copy.to.var=Reply.sum and Assign.allInstances()→exists
              (b,c:Assign|a.copy.from=b.copy.to+c.copy.to))
    }
    inmodel Left{
     context WSBPEL::Assign
      def: lS:Assign=Assign.allInstances()→select(a:Assign|
          a.copy.to.var=Reply.sum and Assign.allInstances()→exists
              (b,c:Assign| a.copy.from=b.copy.to+c.copy.to) and
          let Sum:Real=b.copy.to→value()+c.copy.to→value() in Sum)
    }
    inmodel Right{
     context WSBPEL::Assign
      def: rS:Assign=Assign.allInstances()→select(a:Assign|
          a.copy.to.var=Reply.sum and Assign.allInstances()→exists
              (b,c:Assign|a.copy.from=b.copy.to+c.copy.to) and
          let Sum:Real=b.copy.to→value()+c.copy.to→value() in Sum)
    }
    where{
       Base.bS=Left.lS and Base.bS=Right.rS and Left.Sum< >Right.Sum
    }
    description {
       "Conflict in the value of the sum variable."
    }
}
```

Listing 5: CPL Specification of the Data Flow Inconsistency Conflict

Control Flow Loop. A behavioral semantic conflict that occurs in the UML state machine is an unwanted loop that leads to a deadlock. Figure 8 shows an example of behavioral semantic conflict (adapted from [1]) in which one user adds a simple state *Accessibility* as part of composite state *Evaluation*, while the other user adds

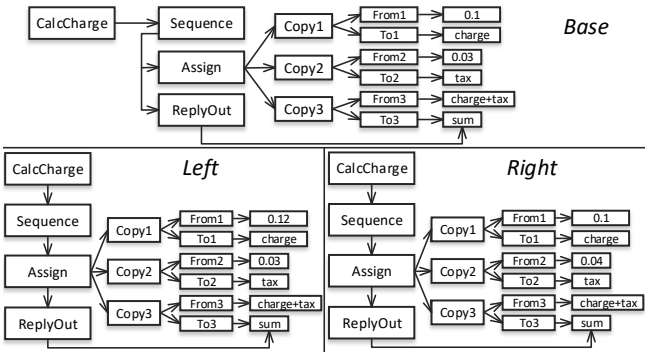

Figure 7: Data Flow Inconsistency: a Behavioral Semantic Conflict

a decision vertex after the join vertex. The integration of these modifications results in a syntactically correct model. However, the runtime execution behavior of the merged model shows that an accessibility error would result in a loop, resulting in a control flow deadlock for the *Launch* state [1].

The control flow loop occurrence is an example of behavioral semantic conflict, which is identified at runtime. This conflict may happen in unknown situations based on the different values of elements at the execution time. Therefore, our formalism is unable to specify the control flow loop by conflict pattern and to express the situation and relationships between elements upon the conflict occurrence.

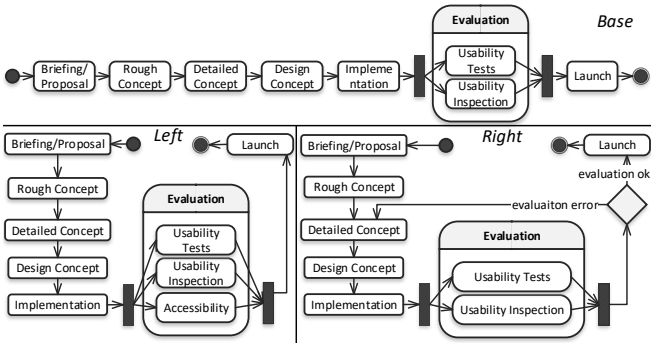

Figure 8: Control Flow Loop: a Behavioral Semantic Conflict

5.2 Discussion

In this section, we structure our discussion to answer the three research questions raised in Section 1.

RQ1) Is it possible to specify any model merging conflict as a pattern?

To answer the first question, we used conflict patterns to specify at least one example for each conflict category described in Section 2.2. In this context, we could specify 4 of 5 merge conflicts. Indeed, conflict patterns cannot express behavioral semantic conflicts [7, 29], which rely on the control flow behavior of the system,

and can only be detected at execution time. For the other types of merging conflict, their specification relies on the expressivity of the language that represents model fragments and their relationships: if the language is able to express any fragment of a given model, then the pattern can specify any merge conflict (other than the control flow behavioral semantic ones). Consequently, based on the results of our experiments on specifying conflicts using the proposed template, we cannot claim that all kinds of model merging conflicts can be specified as a pattern. However, for most of them, this is possible, particularly the conflicts that are not dependent on the unknown situations at the execution time.

RQ2) Is the proposed formalism able to represent model merging conflicts?

To answer the second question, we used CPL, the implementation of the proposed formalism, to represent the same 4 of 5 conflicts. CPL relies on OCL queries to represent model fragments, therefore it dependents directly on the expressiveness of OCL [16]. The examples show that our formalism can represent syntactic and semantic conflicts without ambiguities. We strongly believe that the proposed formalism can represent any model merging conflict, other than control flow behavioral semantic conflicts that cannot be specified as a pattern. Consequently, the proposed formalism is appropriately able to represent model merging conflicts.

RQ3) Is the formalism able to support the representation of conflicts for different modeling languages?

To answer the third question, we use our formalism to represent conflict patterns for models conforming to different modeling languages such as UML class diagram, WSBPEL, and Graph diagram. Although the representation of model merging conflicts for only three modeling languages is not enough, we attribute the ability of our formalism with the *domain* and *fragment* declaration parts, which provide access to different elements of any EMF-based models. The proposed formalism is based on the Ecore and user-defined types and its implementation is based on the OCL grammar, which applies to UML and EMF-based models. Consequently, the CPL formalism can be used to represent conflicts in any UML and EMF-based models.

5.3 Threats to Validity

The main threat to the validity of our study is the choice of conflict cases. While the Pull-Up Method example shows the applicability of the CPL formalism, we have selected five more conflict cases for different modeling languages that contain all conflict categories. This demonstrates that the CPL formalism can be used in the specification of different conflicts from different domains. Nevertheless, in future work, we need to conduct additional conflict cases to assess the generalizability of the CPL formalism.

6 RELATED WORK

There exist several research efforts that use formal specifications to detect conflict or check the consistency of models merging [10, 28–30]. In this section we will restrict ourselves to efforts that propose approaches for expressing model merging conflicts or for model pattern matching.

Cicchetti et al. [8] propose a model-based approach to represent conflicts, based on the differences amongst models. Regarding difference models, a conflict model can be defined to specify model merging conflicts as not allowed contemporary matches between parallel model modifications. In contrast to our approach, their conflict model is not able to describe the conflicts that may arise based on the modification side effects, it can only be used to represent some syntactic and static semantic conflicts.

Sharbaf and Zamani [27] propose to use UML profiles for modeling three-way merging conflicts. In their approach, a conflict model consists of three conflict parts, where each part illustrates the conflict conditions by an example of a model version. However, their approach is restricted for specifying three-way merging conflicts on UML models, whereas we propose a comprehensive approach to represent conflicts for UML and EMF-based models, which includes conflicts concerning two or more models.

Brosch et al. [6] propose an UML profile-based approach visualize conflicts on the merged version of the same UML model. Contrary to our work, their approach only represents conflicts in the concrete syntax of UML modeling language, and the conflict specification is not addressed.

Clark [9] presents an extension to OCL for declarative pattern matching for specifying system states. While Clark introduces an *object-expression* that is used to express instances over one model, our proposal focuses on specifying elements and conditions over multiple models that express a conflict occurrence.

Based on Clark's proposal, Jouault et al. [13] propose an OCL extension called OCLT, which uses pattern matching to enable the declaration of functional model transformations. Similarly to our approach, OCLT also uses OCL for expressing model fragments, however for a different purpose.

Kolovos and Paige [15] present the Epsilon pattern language (EPL), which is a textual language to express the specification of structural patterns on models that conform to arbitrary metamodels. EPL has been implemented on the Epsilon platform [21], which leads to benefit from EOL as a model querying language. EPL supports the pattern matching algorithm to detect specified patterns on multiple models. While EPL is a generic pattern language that works on the Epsilon platform, CPL focuses on model merging conflict specification for all Eclipse-based modeling frameworks. Epsilon could be an alternative to OCL in our work, although less popular to OCL in the modeling community.

7 CONCLUSION AND FUTURE WORK

Concurrent and contradicting changes across collaborative software model evolution in the project lifecycle lead to inconsistencies in software systems. Model merging conflicts are inevitable, rising conflict management to an essential role in collaborative development. However, there is no consensus on a representation approach to specifying a model-merging conflict. To address this problem, we presented the Conflict Pattern Language (CPL), a textual language built upon a well-formed syntax formalism for conflict specification, in terms of examples, syntax definitions, parser, and syntax-aware editor. Using CPL, language engineers are able to formally specify model merging conflicts. The conflict specification can be used to

automatically express the conflict situaitons in the conflict detection and resolution engines for different model merging tools.

CPL helps the slow and error-prone task of specifying and syntactically validating model-merging conflicts by suggesting options while writing conflict specifications. It provides a structural conflict pattern template for specifying model-merging syntactic and semantic conflicts for any EMF-based model. CPL supports model-merging techniques that include N concurrent model versions by allowing unlimited fragments for a conflict pattern. It is based on OCL grammar and tooled by an editor and a parser.

To show the applicability of CPL, we defined a case study, including conflict cases from five different conflict categories. The results showed how CPL could represent different syntactic and semantic conflicts.

As future work, we will investigate further conflict cases to evaluate the generalizability of the CPL formalism. The complementary work is required to implement the semantics of conflict patterns and use them to detect model merging conflicts. Furthermore, we intend to extend the textual editor of CPL by developing a graphical editor to help the specification of conflicts. Finally, we further plan to conduct a systematic comparative analysis to investigate the required effort for the specification of conflicts.

REFERENCES

[1] Kerstin Altmanninger and Alfonso Pierantonio. 2011. A categorization for conflicts in model versioning. *e & i Elektrotechnik und Informationstechnik* 128, 11-12 (2011), 421–426.

[2] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. 2009. A survey on model versioning approaches. *International Journal of Web Information Systems*. 5, 3 (2009), 271–304. https://doi.org/10.1108/17440080910983556

[3] Ian Bayley and Hong Zhu. 2010. Formal specification of the variants and behavioural features of design patterns. *Journal of Systems and Software* 83, 2 (2010), 209–221. https://doi.org/10.1016/j.jss.2009.09.039

[4] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

[5] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. 2012. An introduction to model versioning. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 336–398. https://doi.org/10.1007/978-3-642-30982-3_10

[6] Petra Brosch, Horst Kargl, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti Kappel. 2010. Conflicts as first-class entities: a UML profile for model versioning. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 184–193. https://doi.org/10.1007/978-3-642-21210-9_18

[7] Petra Brosch, Martina Seidl, and Magdalena Widl. 2013. Semantics-Aware Versioning Challenge: Merging Sequence Diagrams along with State Machine Diagrams. *Softwaretechnik-Trends* 33, 2 (2013), 84–86.

[8] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. 2008. Managing model conflicts in distributed development. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 311–325. https://doi.org/10.1007/978-3-540-87875-9_23

[9] Tony Clark. 2013. OCL Pattern Matching. In *OCL Workshop @ MoDELS, ser. CEUR Workshop Proceedings, vol. 1092*. CEUR-WS.org, 33–42. http://ceur-ws.org/Vol-1092/clark.pdf

[10] Hoa Khanh Dam, Alexander Egyed, Michael Winikoff, Alexander Reder, and Roberto E. Lopez-Herrejon. 2016. Consistent merging of model versions. *Journal of Systems and Software*. 112, 1 (feb 2016), 137–155. https://doi.org/10.1016/j.jss.2015.06.044

[11] Mirco Franzago, Davide Di Ruscio, Ivano Malavolta, and Henry Muccini. 2017. Collaborative model-driven software engineering: a classification framework and a research map. *IEEE Transactions on Software Engineering* 44, 12 (2017), 1146–1175.

[12] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, et al. 2007. Web services business process execution language version 2.0. *OASIS standard* 11, 120 (2007), 5.

[13] Frédéric Jouault, Olivier Beaudoux, Matthias Brun, Mickael Clavreul, and Guillaume Savaton. 2015. Towards functional model transformations with ocl. In *International Conference on Theory and Practice of Model Transformations*. Springer, 111–120.

[14] Nirmal Kanagasabai, Omar Alam, and Jörg Kienzle. 2018. Towards online collaborative multi-view modelling. In *International Conference on System Analysis and Modeling*. Springer, 202–218.

[15] Dimitris S Kolovos and Richard F Paige. 2017. The epsilon pattern language. In *International Workshop on Modelling in Software Engineering (MiSE)*. IEEE, 54–60. https://doi.org/10.1109/MiSE.2017.8

[16] Luis Mandel and María Victoria Cengarle. 1999. On the Expressive Power of OCL. In *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I (Lecture Notes in Computer Science)*, Jeannette M. Wing, Jim Woodcock, and Jim Davies (Eds.), Vol. 1708. Springer, 854–874. https://doi.org/10.1007/3-540-48119-2_47

[17] Constantin Masson, Jonathan Corley, and Eugene Syriani. 2017. Feature Model for Collaborative Modeling Environments.. In *MODELS (Satellite Events)*. 164–173.

[18] T. Mens. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*. 28, 5 (may 2002), 449–462. https://doi.org/10.1109/TSE.2002.1000449

[19] Tom Mens, Gabriele Taentzer, and Olga Runge. 2005. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electronic Notes in Theoretical Computer Science* 127, 3 (April 2005), 113–128. https://doi.org/10.1016/j.entcs.2004.08.038

[20] Object Management Group. 2014. Object Constraint Language (OCL) Specification. Version 2.4. https://www.omg.org/spec/OCL/2.4/

[21] Richard F Paige, Dimitrios S Kolovos, Louis M Rose, Nicholas Drivalos, and Fiona A.C. Polack. 2009. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering *(14th IEEE International Conference on Engineering of Complex Computer Systems)*. IEEE, 162–171. https://doi.org/10.1109/ICECCS.2009.14

[22] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2018. Collaborative modeling and group decision making using chatbots in social networks. *IEEE Software* 35, 6 (2018), 48–54.

[23] Hendrik Radke, Thorsten Arendt, Jan Steffen Becker, Annegret Habel, and Gabriele Taentzer. 2018. Translating essential OCL invariants to nested graph constraints for generating instances of meta-models. *Science of Computer Programming* 152 (2018), 38–62.

[24] Sudha Ram and V. Ramesh. 1998. Collaborative Conceptual Schema Design: A Process Model and Prototype System. *ACM Trans. Inf. Syst.* 16, 4 (Oct. 1998), 347–371. https://doi.org/10.1145/291128.291130

[25] Alireza Rouhi and Bahman Zamani. 2016. Towards a formal model of patterns and pattern languages. *Information and Software Technology* 79 (2016), 1–16. https://doi.org/10.1016/j.infsof.2016.06.002

[26] Firoozeh Sepehr and Donatello Materassi. 2019. Blind Learning of Tree Network Topologies in the Presence of Hidden Nodes. *IEEE Trans. Automat. Control* (2019).

[27] Mohammadreza Sharbaf and Bahman Zamani. 2017. A UML profile for modeling the conflicts in model merging. In *International Conference on Knowledge-Based Engineering and Innovation (KBEI)*. IEEE, 0197–0202. https://doi.org/10.1109/KBEI.2017.8324972

[28] Mohammadreza Sharbaf and Bahman Zamani. 2020. Configurable Three-way Model Merging. *Software: Practice and Experience* 50, 8 (2020), 1565–1599. https://doi.org/10.1002/spe.2835

[29] Mohammadreza Sharbaf, Bahman Zamani, and Behrouz Tork Ladani. 2015. Towards automatic generation of formal specifications for UML consistency verification. In *International Conference on Knowledge-Based Engineering and Innovation (KBEI)*. IEEE, 860–865. https://doi.org/10.1109/KBEI.2015.7436156

[30] Gerson Sunyé. 2017. Model Consistency for Distributed Collaborative Modeling. In *Modelling Foundations and Applications - 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings (Lecture Notes in Computer Science)*, Anthony Anjorin and Huáscar Espinoza (Eds.), Vol. 10376. Springer International Publishing, Cham, 197–212. https://doi.org/10.1007/978-3-319-61482-3_12

[31] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. 2001. Refactoring UML Models. In *«UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings (Lecture Notes in Computer Science)*, Martin Gogolla and Cris Kobryn (Eds.), Vol. 2185. Springer, 134–148. https://doi.org/10.1007/3-540-45441-1_11

[32] Jim Whitehead. 2007. Collaboration in software engineering: A roadmap. In *Future of Software Engineering (FOSE'07)*. IEEE, 214–225.

[33] Hong Zhu. 2010. On the theoretical foundation of meta-modelling in graphically extended BNF and first order logic. In *International Symposium on Theoretical Aspects of Software Engineering*. IEEE, 95–104. https://doi.org/10.1109/TASE.2010.11

[34] Hong Zhu and Lijun Shan. 2006. Well-formedness, consistency and completeness of graphic models. In *9th UKSim-AMSS International Conference on Computer Modelling and Simulation, UKSim 2006, Oriel College, Oxford, United Kingdom, 4-6 April 2006*. 47–53.