Software and Systems Modeling (2022) 21:2461–2487
https://doi.org/10.1007/s10270-022-00976-4

REGULAR PAPER

An efficient line-based approach for resolving merge conflicts in XMI-based models

Alfonso de la Vega[1] · **Dimitris Kolovos**[2]

Received: 8 July 2021 / Revised: 24 November 2021 / Accepted: 10 January 2022 / Published online: 9 March 2022

Abstract
Conflicts in software artefacts can appear during collaborative development through version control systems. When these conflicts happen in XMI models, the conflict sections generated by diff programs break the XMI serialisation and compromise the ability to use model editors that assume well-formedness of this serialisation. While these conflict sections already mark the conflicting lines of the model, current tools for conflict resolution in models ignore them and instead load the different versions of a model from the repository, over which they perform a full and costly comparison that re-identifies the conflicts. We present a novel approach that prevents this repetition of work by directly parsing XMI-based models with conflict sections, which allows for a targeted analysis of only the lines of the model that have been detected to be in conflict by the version control system. We have implemented this approach in the PEACEMAKER tool, which can load XMI models with conflict sections, compute and display conflicts at the model level, and provide appropriate resolution actions. Compared with state-of-the-art model comparison tools with support for conflict resolution, PEACEMAKER is able to identify the vast majority of conflicts in models while reducing the required time by up to 60%. The small subset of non-identified conflicts does not introduce issues into the models, e.g. there is no loss of model information, and the resulting models after line-merging these conflicts conform to their metamodels.

Keywords Model-driven engineering · Version control systems · Conflict resolution

1 Introduction

Software development is usually a collaborative endeavour, and it is frequent to find several developers working on different aspects of the same software system at the same time. Version control systems (VCSs) make concurrent work possible, by allowing developers to work on different versions or *branches* of the codebase.

Nevertheless, concurrent work can cause conflicts to appear when merging two branches where incompatible changes have been made. This work focuses on file-based VCS, such as Git or Subversion. In this type of VCSs merge operations take place at a file-line level, by performing a three-way comparison between the two branches being merged and their common ancestor in the version tree [1,2]. As a conflict example, when the same line of code has been modified in different ways by two branches, the VCS does not know which line version should be selected, so a conflict is raised for the developer to resolve.

In model-driven software development, models become additional software artefacts to manage during their evolution [3,4]. While there are several model storage solutions for model versioning and persistence [5–8], it is also common (if not the norm) to find models stored in mainstream file-based VCSs alongside related source code. This work focuses on the latter, specifically on EMF models persisted in the standard XMI format [9] and versioned in Git repositories.

XMI models stored in VCSs can suffer from the same issues as any other versioned artefacts, including merge

Communicated by Philippe Collet.

The work presented in this paper has been funded through the HICLASS InnovateUK project (Contract No. 113213).

✉ Alfonso de la Vega
alfonso.delavega@unican.es

Dimitris Kolovos
dimitris.kolovos@york.ac.uk

[1] Software Engineering and Real-Time, Universidad de Cantabria, Santander, Spain

[2] Department of Computer Science, University of York, York, UK

conflicts. When conflicts appear in models, though, developers find themselves at a disadvantage in comparison with how conflicts in regular source files are managed. Conflicting lines are surrounded by the VCS with special delimiters, i.e. <<<<, ====, or >>>>. For source files in a human-readable textual format, these delimiters might prevent successful compilation, but a developer can open these files with the text editor or IDE of their choice, which could even apply a special highlight to the conflicts to ease their resolution. Nevertheless, current EMF-based editors, such as EMF's reflective tree-based editor [5] or Sirius [10], require well-formed XMI models as input to be able to open them. So, any presence of conflict delimiters causes these editors to fail to even parse the models. Also, the alternative of resolving conflicts in models by directly modifying XMI in a text editor is a tedious and error-prone task.

As regular model editors do not work in such cases, developers have to resort to special tools to resolve conflicts, such as EMF Compare [11] or EMF DiffMerge [12]. However, these tools are also unable to parse model files where the conflicting lines have already been marked by the VCS. Instead, all three versions of the models in conflict (i.e. the two versions being merged and their common ancestor) are loaded from the VCS, and a full three-way match-and-compare operation is performed at model level to re-identify the conflicts. Therefore, a lot of work that had already been done by the VCS is repeated. Once identified, conflicts are presented in a special editor for their resolution. The work of resolving these conflicts has to be done in a single session, as existing tools do not allow saving partial progress if there are conflicts remaining in the model.

In this work, we aimed to answer the following research questions:

- *RQ*1 Can an approach that uses the line-based conflicts marked by a VCS to identify conflicts instead of a full model comparison offer the same results?
- *RQ*2 Does a conflict detection approach as described in *RQ1* provide better performance and scalability than full model comparison?
- *RQ*3 Is relying on line-based VCSs such as Git to merge XMI models and to detect conflicts safe? More specifically, we wish to examine whether text-level merge operations can produce conflict-free but inconsistent or otherwise undesirable XMI models.

RQ1 and *RQ2* came from our interest in leveraging the work that is already done by the VCS, instead on having to re-identify the conflicts via comparing the full model versions. *RQ3* is a more general question that derives from *RQ1*. For instance, if the line-based conflict detection—as implemented in contemporary file-based VCS—cannot detect some types of conflicts in models, and this is the only way in which conflicts are sought (e.g. when using Git from a command-line interface), then some conflicts might end up undetected and cause problems later. Previous works already discourage the detection of conflicts in models using line-based merging [13–15], because this method is not able to correctly identify some changes as conflicting or not. For instance, new (but unrelated) model elements that end up in the same XMI line of the two model versions cause a false positive conflict when line-merged. Also, moving model elements to a different position in the model can cause duplications of these elements when performing a line-based merge, and this duplication will not be detected by the VCS. Despite the existing research, we wanted to provide a detailed analysis on the issue, and to determine if there is a set of constraints under which merging XMI models at line level can be made safe.

We present a novel conflict resolution approach that is able to parse XMI-based models in which a VCS has already identified conflicting lines. By performing a targeted analysis of these lines, and consequently to the model elements contained in them, our approach can save a lot of unnecessary work during the conflict identification process. In addition, this approach is able to detect issues that can appear when doing a line-based merge of XMI models, such as model element duplications. We have implemented the approach in PEACEMAKER, which is an Eclipse-based tool that provides an editor for visualising and resolving the identified conflicts at the model level.

We compared the completeness and performance of PEACEMAKER with two state-of-the-art tools for model comparison that also allow resolving conflicts (EMF Compare and EMF DiffMerge). Our results indicate that PEACEMAKER is able to detect the vast majority of (but not all) conflicts, while taking up to $\sim$ 60% less time for that detection. In those types of conflicts that are missed by PEACEMAKER, no issues are introduced into the models, e.g. after carrying out a line-based merge of those conflicts there is no loss of model information, and the resulting models conform to their metamodels. We also include a discussion on how Git can be used to safely version XMI models. Our conclusion is that, in the general case, it is not safe to merge XMI models using plain line-based Git merge operations, as they can produce structurally consistent but semantically incorrect models. Consequently, we consider using custom merge strategies when working with models in line-based VCSs a mandatory requirement. Also, if the small subset of missed conflicts can be tolerated, we believe PEACEMAKER offers a considerable performance improvement over existing model comparison and merge tools. This improvement might be useful for those contexts where engineers work with large and frequently updated models with clearly assigned responsibilities over different parts of the model.

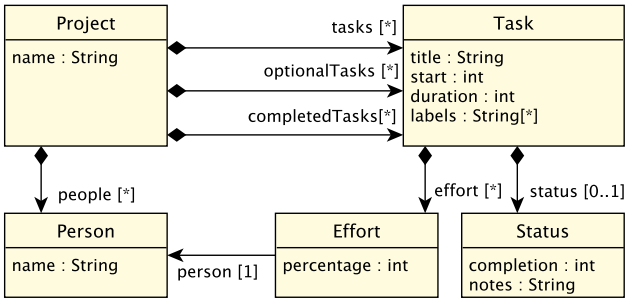

Fig. 1 Project Scheduling Language (PSL) metamodel in Ecore

The rest of this paper is structured as follows: Sect. 2 introduces a running example and some relevant background. Section 3 details the line-based approach to process models with conflicts, Sect. 4 presents the implemented PEACEMAKER software components, and Sect. 5 compares our approach with existing state-of-the-art tools. Finally, Sect. 6 discusses related work, and Sect. 7 concludes the article and outlines future work.

2 Background

2.1 Running example

We use a contrived project scheduling language (or *PSL*) to depict scenarios with models in conflict. Figure 1 shows an Ecore metamodel including the main elements of the language. In PSL, a *Project* is composed of *Tasks*, which have a title, a start time, a duration, and a list of labels. There are also *optionalTasks* which are stored separated from standard *tasks*. Also, when a task is completed, it is moved to the *completedTasks* list.[1] Tasks are carried out by *Persons* that contribute an *Effort* (percentage of their time) for the completion of each task in which they participate. Lastly, tasks can have a *Status* that includes its completion percentage and some optional notes.

2.2 Model serialisation

In this work, we are concerned with models persisted in the standard XML Metadata Interchange (XMI) format [9]. XMI is a standard developed by the Object Management Group that offers XML schemas for the storage and exchange of models based in the Meta-Object Facility standard [16]. XMI is the default model serialisation format of EMF, which is currently the most widely used open-source domain-specific modelling framework.

[1] While the "completed" and "optional" tasks could be specified more appropriately (e.g. by a derived feature over the *completion* attribute of a task's *status*, and by an *optional* Boolean attribute), we use three containment references to be able to illustrate a specific type of conflict, denoted Containing Feature Update (described in Sect. 3.3).

```
1  <Project xmi:id="project1" name="Blog">
2    <tasks xmi:id="task1"
3           title="Requirements" duration="2">
4      <effort xmi:id="e1" person="aliceID"/>
5    </tasks>
6    <people xmi:id="aliceID" name="Alice"/>
7  </Project>
```

Listing 1 PSL model serialised in XMI

Listing 1 contains a PSL model expressed in XMI that conforms to the metamodel of Fig. 1. In XMI, an XML tag marks the start of an element belonging to the serialised model. For instance, line 1 contains the starting tag of a *Project* element, while line 2 starts a *Task*. In the context of EMF models, the starting tag of an XML element contains all attribute values of the corresponding model element. In the example of Listing 1, the *Project* tag includes a *name*, which is an EAttribute of the corresponding type in the PSL metamodel of Fig. 1. Starting tags also store the value of non-containment references. As an example, line 3 defines an *Effort* element, and the tag contains a *person* EReference attribute that points to the id of the person who will contribute this effort.

In addition to the XML attributes storing element features, special attributes can appear to include external metadata into the elements. The name of these attributes is typically prefixed by *xmlns* (used to declare XML namespaces) or *xmi* (allows indicating special features related to XMI serialisation). The most relevant special attribute for this work is *xmi:id*, which can be used to indicate an extrinsic (i.e. not part of the element features) and immutable identifier for a model element. For the sake of simplicity and the better use of space in figures, in the remaining of the paper any occurrence of an *id* attribute inside an XMI element refers to *xmi:id*.

The remaining features of EMF model elements, i.e. containment references, are represented by the nesting of XML elements in the XMI representation of the model. For instance, *Project* is the root element of Listing 1 starting at line 1. This element contains a task that starts in line 2, and a person in line 5. Both these nested elements are part of the project's *tasks* and *people* containment references, respectively. In those elements, the name of the containment reference is used as tag name to allow having several containment references of the same type (e.g. *Person, Task*) in the same model element.

2.3 Syntax of conflict sections

We describe now the syntax that is used by contemporary file-based VCSs to mark lines after a merge with conflicts.

In this work, we used Git for model versioning; however, as the syntax of conflict sections and the line-based operations used by the different VCSs is practically identical [2,17,18], these descriptions and the results of our work apply to other VCSs too (e.g. Subversion).

In its simplest form, a conflict section consists of two segments: one containing the changes of the left version (usually the local one); and other corresponding to the right version (generally the version in the repository index). These segments are surrounded by specific conflict section separators: <<<<, ====, and >>>>. Nevertheless, it is possible to include extra information in the conflict section: the *diff3* conflict style allows adding an extra segment to the generated conflict sections. An extra separator (||||) is used to include the ancestor segment after the left version one. This segment includes the contents of the common ancestor of the left and right branches that is used in the three-way merge, and it allows for a better analysis of the causes behind the appearance of conflicts [1].

Listing 2 shows a model with two conflict sections, each one composed of a left, ancestor, and right segments as described in the previous paragraph. The first conflict section goes from line 5 to 11, while the second one starts at line 15 and ends at line 23. The changes that have caused these conflicts are the following:

- The first conflict section is caused by an update of the start and duration times of the *Development* task. The left version updates both start and duration (line 6), while the right one only modifies the duration (line 10).
- The second conflict section is caused by an update of the *Deployment* task effort. The ancestor information shows that, originally, the effort for this task was split evenly (i.e. 50:50) between Alice and Bob (lines 19–20). Nevertheless, while the left version adjusted the original efforts of Alice and Bob from 50:50 to 70:30 (lines 16–17), the right version removed Bob from this task and assigned Alice the full effort (line 22). A percentage is not present in this last effort because 100% is the default value for that attribute.

2.4 Conflict detection through model comparison

We now introduce how state-of-the-art model comparison tools identify conflicts in XMI-based models. Instead of using merged models such as the one of Listing 2, these tools find conflicts by performing a three-way comparison between the left, ancestor and right model versions, which are gathered from the VCS. EMF Compare [11] and EMF DiffMerge [12] are two of the most mature tools for the comparison of EMF models, and both of them are actively maintained. The descriptions of this section apply to both of these tools.

Model versions are processed through a multi-stage comparison pipeline. Despite differences between tool pipelines, there are at least two stages that are performed in every case. There is a *matching* stage where model elements from the different versions are matched in preparation for subsequent comparison. The most precise way to match model elements is by means of unique identifiers such as XMI ids, but in absence of these it is also possible to apply similarity/distance techniques [19]. Once the model element versions have been matched, they are compared in a *diff* stage to find all changes.

When looking for conflicts, an extra step needs to be performed to detect incompatible changes between the model versions. For instance, if both the left and right version have changed the value of an EAttribute with respect to the ancestor version, these changes are incompatible, and as such they are marked as a conflict for the developer to resolve.

3 Processing XMI models with conflicts

PEACEMAKER is able to parse and load EMF models serialised in XMI that contain conflict sections, with the requirement of model elements having unique identifiers. These identifiers are used for matching the model elements across the two versions. XMI offers the possibility to set an extrinsic id using the special *xmi:id* XML attribute. Alternatively, Ecore allows marking an EAttribute of each EClass by setting its *id* attribute to true, which indicates that the value of the marked EAttribute should be used as the identifier for instances of its container EClass.

The following sections describe the steps involved in processing the model of Listing 2.

3.1 Line-based preprocessing

The first step involves a preprocessing of the model lines to detect any conflict sections. This preprocessing consists of two tasks:

1. *Line identification* Each line of the input model is given a type that indicates if it is a *common* line (i.e. not contained in a conflict section), a *separator* (e.g. <<<<), or if instead it belongs to the *left*, *ancestor* or *right* segments of a conflict section. Also, existing conflict sections are identified and related to their file lines for later. Figure 2a, b shows the line type assignments and the identified conflict sections for Listing 2, respectively.
2. *Extraction of model versions* This step extracts the left, ancestor and right versions of the model by selecting the respective line types identified during the previous step. Figure 2c shows which lines of the original model of Listing 2 belong to each version. For instance, the left

```
1  <Project xmi:id="project1" name="Blog">
2    <tasks xmi:id="task1" title="Requirements" duration="2">
3      <effort xmi:id="e1" person="aliceID"/>
4    </tasks>
5  <<<<<<< left.model
6    <tasks xmi:id="task2" title="Development" start="3" duration="3">
7  ||||||| base.model
8    <tasks xmi:id="task2" title="Development" start="2" duration="2">
9  =======
10   <tasks xmi:id="task2" title="Development" start="2" duration="1">
11 >>>>>>> right.model
12     <effort xmi:id="e2" person="bobID"/>
13   </tasks>
14   <tasks xmi:id="task3" title="Deployment" start="4" duration="1">
15 <<<<<<< left.model
16     <effort xmi:id="e3" person="aliceID" percentage="70"/>
17     <effort xmi:id="e4" person="bobID" percentage="30"/>
18 ||||||| base.model
19     <effort xmi:id="e3" person="aliceID" percentage="50"/>
20     <effort xmi:id="e4" person="bobID" percentage="50"/>
21 =======
22     <effort xmi:id="e3" person="aliceID"/>
23 >>>>>>> right.model
24   </tasks>
25   <people xmi:id="aliceID" name="Alice"/>
26   <people xmi:id="bobID" name="Bob"/>
27 </Project>
```

Listing 2 XMI notation of a PSL model with conflicts

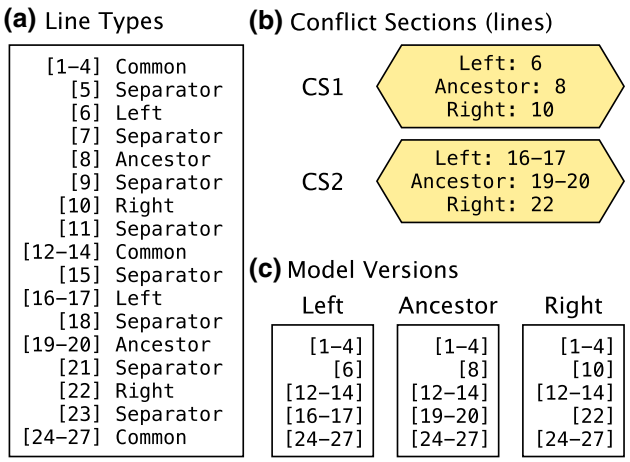

Fig. 2 **a** Identified types for the lines of Listing 2. **b** Detected conflict sections in the model. **c** Lines of the original model that belong to each extracted model version

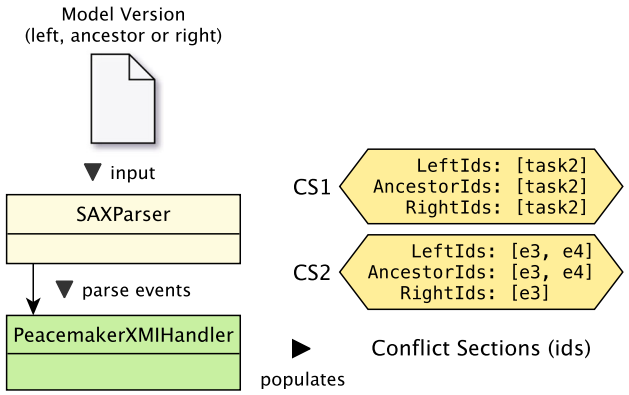

Fig. 3 Conflict sections are populated with the contained model element identifiers during the loading of model versions

version of the model would be composed of *common* and *left* lines, these are, 1–4, 6, 12–14, 16–17, and 24–27.

3.2 Parsing of model versions

The previous step provides model versions that are free from the conflict section separators that originally broke the serialised XMI notation. As a result, these versions can be parsed and loaded as if they were standard XMI, which is done during this second step.

Figure 3 contains the parts of the XMI parsing and loading process that are relevant for this work. In the context of EMF, XMI parsing is carried out using an event-based SAX parser [20]. The EMF XMI parser processes XMI models sequentially, this is, elements are read by a parser one by one from the input model, and then events that indicate the appearance of these elements are triggered and processed by an event handler. In PEACEMAKER, we created a custom event handler (denoted as *PeacemakerXMIHandler* in the figure) that extends the default one. Apart from carrying out its model

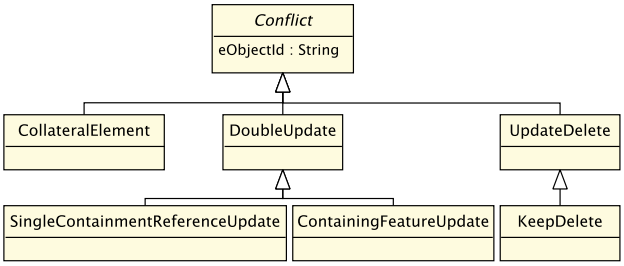

Fig. 4 Conflict types found in conflict sections

loading duties, this handler also checks the lines of the elements being loaded against the lines belonging to the conflict sections identified during the preprocessing step. This check is done against the conflict section segment that matches the model version being loaded, i.e. left, ancestor, or right. When the model element lines are found inside the associated segment for a concrete version, the element identifier is registered in the conflict section, again in the appropriate list of identifiers. After loading the three model versions, the identifiers of the actual model elements that are part of a conflict section have been captured, instead of just knowing their lines.

Let us assume we are loading the left version of the model of Listing 2. The first task element that is found is the *Requirements* task. When the loading of this element starts, the position of the line that contains its starting tag is checked by the event handler against the conflict sections detected in the preprocessing step. As line 2 is not part of any of the conflict sections, this element is not relevant for the detection of conflicts, so nothing is marked. The second task element is *Development*. The contents of this element can be traced back to line 6 of the original model, which belongs to the left segment of the first conflict section. Therefore, the id of this task, *task2*, is added to the list of left identifiers of the conflict section. Similar processing takes place for the remaining elements of the three model versions. The resulting conflict sections populated with conflicting identifiers can be seen on the right of Fig. 3.

3.3 Conflict section analysis

Once the conflict sections have been populated with the identifiers of the elements in conflict, they can be analysed to detect which type of conflicts are present in the original model.

Figure 4 shows a categorisation of the different conflicts that can take place in a conflict section. The *Conflict* abstract class on top stores common features of all concrete conflicts. For instance, all conflicts happen around a model element identifier (*eObjectId*).

Table 1 Conflict type that applies depending on the combination of identifiers found in the segments of a conflict section

Conflict Section Segment			Potential Conflict
Left	Ancestor	Right	
ID_1	ID_1	ID_1	Double Update, Containing Feature Update
ID_1		ID_2	Single Containment - Reference Update
ID_1	ID_1		Update Delete, Keep Delete
	ID_1	ID_1	
ID_1			Collateral Element
		ID_1	

Conflicts are detected based on the combination of model element identifiers found in the segments of a conflict section. This combination determines which type of the conflicts of Fig. 4 applies on each case, as indicated in Table 1. We now describe these conflict types, including possible actions that can be taken to resolve them. These descriptions are supported by the examples of Fig. 5.

- *Double Update* A model element with the same identifier (e.g. ID_1 in Table 1) has been updated in both the left and right versions. Precisely, this conflict appears when either an attribute and/or a non-containment reference of an element have been modified. As resolutions for this type of conflict, PEACEMAKER offers to either keep the left or right version of the element. Figure 5a shows a Double Update conflict where the title of the *t1* task has been updated in the left and right versions. PEACEMAKER can detect if versions change the same element, but in non-conflicting ways (e.g. when each version modifies different element features), so that no conflict is detected in the end. This and other Git false positives are discussed later in Sect. 3.4.
- *Containing Feature Update* This conflict happens when the containing feature of an element is changed to two different features in the left and right versions. Again, the user is offered the options to either keep the left or right modifications, i.e. leaving the element under the updated containment feature of the left version, or instead under the feature selected by the right version. In Fig. 5c the *t1* task, initially contained under the *tasks* reference of the *p1* project, is moved to the *optionalTasks* reference on the left, and to the *completedTasks* one on the right.
- *Single Containment Reference Update* This is a special case of the Double Update conflict that takes place when a model element has a containment, single-valued reference which has been updated in the left and right versions. It is also a trickier conflict to detect, because the id of the

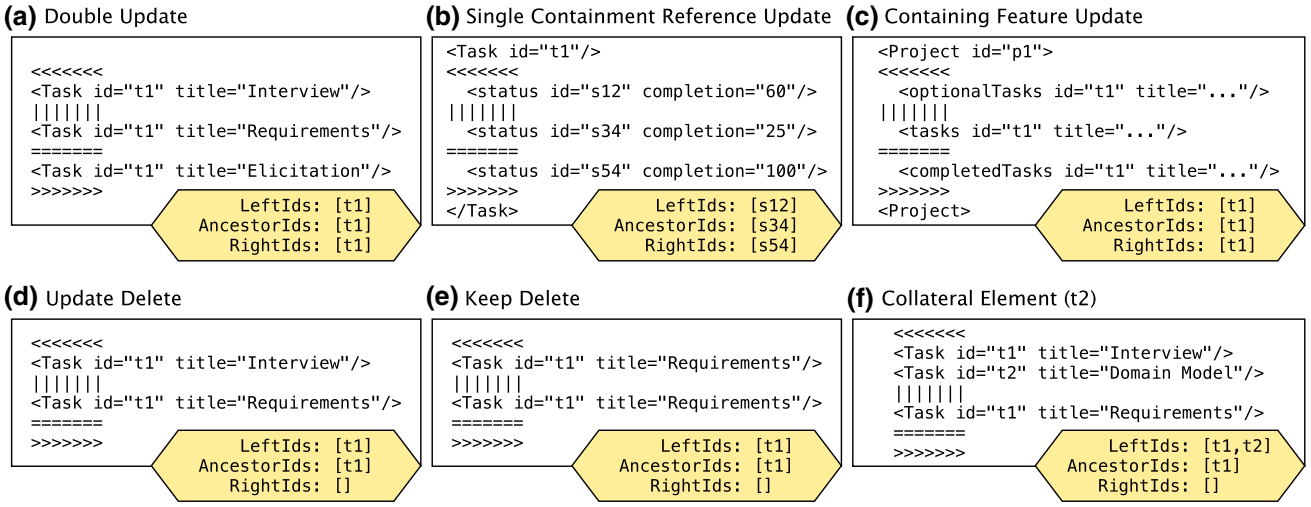

Fig. 5 Examples of the different line conflicts along with the element identifiers in the conflict sections

element contained by the reference is different for both versions of the reference value (e.g. ID_1 and ID_2 in Table 1). If it were the same id, the conflict would be identified as the general Double Update one. Like in the previous conflict, the user is offered the options to keep either the left or right version of the contained element. Figure 5b depicts an example of this conflict where the status element of the *t1* task has been updated to different Status elements: an *s12* element has been set on the left, while *s54* appears on the right.

– *Update Delete* An element with the same id has been updated in one of the versions, but deleted in the other. To identify this conflict, PEACEMAKER checks the ancestor version to see if the element in conflict existed prior to the changes introduced by the left and right versions. If this conflict happens, the user is offered the options to either keep the updated version, or accept the deletion and remove the element. Figure 5d shows a *t1* task that has been updated on the left version (title change), while this task has been deleted on the right. Notice that the *t1* task was also present in the ancestor version.
– *Keep Delete* This is a special Update Delete case where an element is deleted in one version, but kept the same (i.e. unmodified with respect to the ancestor) in the other one. This kind of conflict is usually found when other conflicts are detected in adjacent lines. As the changes only appear in one of the versions (i.e. the removal of an element), this could be treated as a false positive and automatically merged by removing the conflicting element. However, for some scenarios it is useful to also leave the decision to either keep or remove the unmodified element to the end user. We present such an example in Sect. 5.1.3. Figure 5e shows an example that is identical to the d case, but this time no update of the title of the *t1* task was carried out in the left version.
– *Collateral Element* This case happens when an element from a conflict section is not found in conflict with any other element. One way this could happen is as a result of finding a conflicting line during the merge, and then having subsequent lines in either of the segments of a conflict section defining new elements that are not in conflict, but that get automatically included as part of the conflict section. Figure 5f shows an example of this, where the same Update Delete conflict shown in case d takes place, but in addition a collateral *t2* task appears in the left version, after *t1*. This can be resolved by either keeping or removing the collateral element. If desired, an automatic resolution can be applied to omit this conflict type and always keep these elements.

The complete analysis of the conflict section identifiers that results in the identification of conflicts is shown in Algorithm 1. This algorithm completes the information of Table 1, showing the extra steps to e.g. determine if a Double Update conflict is of type Containing Feature Update (lines 6–10) or Single Bounded Reference Update (lines 12–16), or which type of Update Delete conflict applies (lines 17–23 and 29–35).

Continuing with the example of Listing 2, and based on the information of Fig. 3(right), the first conflict section contains a Double Update conflict, because both the left and right list of identifiers contain the *task2* id, i.e. the identifier of the *Deployment* task. In the second conflict section, a Double Update and an Update Delete conflicts are found. The Double Update one is caused by the presence of Alice's *e3* effort id in both the left and right identifier lists, while the Update Delete conflict happens because Bob's *e4* effort is present in the lists of left and ancestor ids, while it is not present in the list of right ids.

Input: Conflict Section ($LeftIds$, $RightIds$, $AncestorIds$)
Input: $getLeft(id)$, $getRight(id)$, and $getAncestor(id)$: get model elements by identifier from a model version
Output: A set of conflicts C

```
1  C ← ∅;
2  foreach leftId ∈ LeftIds do
3  |  leftElem ← getLeft(leftId);
4  |  if leftId ∈ RightIds then
5  |  |  rightElem ← getRight(leftId);
6  |  |  if leftElem.containingFeature ≠ rightElem.containingFeature then
7  |  |  |  C ← C ∪ {ContainingFeatureUpdate(leftId)};
8  |  |  else
9  |  |  |  C ← C ∪ {DoubleUpdate(leftId)};
10 |  |  end
11 |  |  RightIds ← RightIds − {leftId};
12 |  else if leftElem.containingFeature is single and containment and
13 |  |  there exists a right element rightElem so that
14 |  |  |  rightElem = getRight(leftElem.container.id) and
15 |  |  |  rightElem.get(leftElem.containingFeature).id ≠ leftId then
16 |  |  C ← C ∪ {SingleContainmentReferenceUpdate(leftElem.container.id, leftElem.containingFeature)};
17 |  else if leftId ∈ AncestorIds then
18 |  |  if equals(leftElem, getAncestor(leftId)) then
19 |  |  |  C ← C ∪ {KeepDelete(leftId)};
20 |  |  else
21 |  |  |  C ← C ∪ {UpdateDelete(leftId)};
22 |  |  end
23 |  else
24 |  |  C ← C ∪ {CollateralElement(leftId)};
25 |  end
26 end
   // There might be remaining identifiers in rightIds
27 foreach rightId ∈ rightIds do
28 |  rightElem ← getRight(rightId);
29 |  if rightId ∈ AncestorIds then
30 |  |  if equals(rightElem, getAncestor(rightId)) then
31 |  |  |  C ← C ∪ {KeepDelete(rightId)};
32 |  |  else
33 |  |  |  C ← C ∪ {UpdateDelete(rightId)};
34 |  |  end
35 |  else
36 |  |  C ← C ∪ {CollateralElement(rightId)};
37 |  end
38 end
```

Algorithm 1: Procedure to identify the kind of conflicts contained in a conflict section

The application of a resolution action updates the contents of the in-memory model versions that were loaded in the parsing step (see Sect. 3.2) according to the selected action. For instance, the Double Update conflict identified around the tasks with id *task2* in the example of Listing 2 could be resolved by keeping the left or right versions. If keeping right were selected, the element of the right model version with id *task2* would replace the one with the same id in the left model version, removing the discrepancy and thus resolving the conflict.

3.4 Detecting git false positives

In some cases, Git wrongly marks model elements as in conflict because of minor differences in their XMI serialisation, or due to unrelated changes clashing into the same lines of the model file. False positives could also be caused by end users directly editing the raw XMI and including minor inconsistencies, such as indentation or style differences. However, manual editing of XMI files is out of the scope of this work, as we assume models are modified in editors that persist changes through an automated serialiser.

Figure 6 shows one of the false positive examples that Git can misreport. In the example, there is a real Update Delete conflict over the *s1* Status element, which is updated in the left version while deleted in the right one (similar to Fig. 5d). The false positive is caused by a minimal change in the parent element that contains the deleted status, i.e. the *t1* task. When the deletion of a model element leaves the parent element without any contents, a self-closing tag is used to persist the

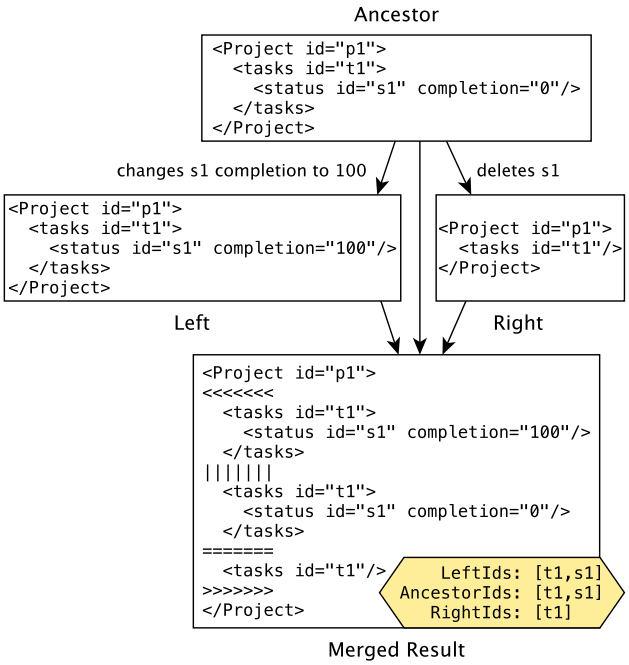

Fig. 6 Git undesirably marks as a conflict the ending tag change of task *t1* when all its contents (i.e. *s1*) are deleted

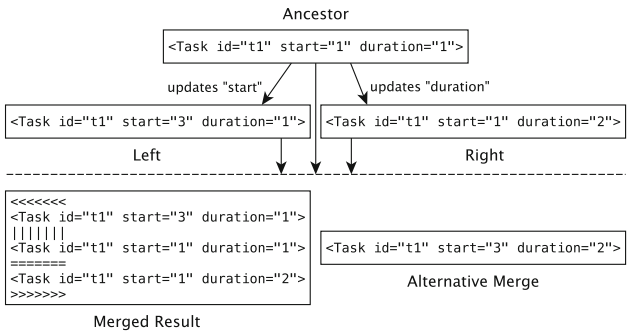

Fig. 7 Git detects as a conflict two simultaneous updates of the *t1* task that affect different features (*start* on the left, and *duration* on the right). An alternative merge is shown at the bottom right, where the disjoint changes are combined

element in XMI, this is, the element is persisted with a single XML tag ending in /> instead of using > along with a closing </task> tag. This self-closing tag is used for the *t1* task on the right version, as it has no extra contained elements after deleting the *s1* status. That subtle change makes Git to detect the line in conflict, and thus to include the model element into the conflict section, as can be seen at the bottom of the figure. PEACEMAKER can detect these false positives by checking the value of the element features between the versions, so in the end only those elements with real conflicts are reported.

The following case is also a source of false positives. A line-based merge reports a conflict every time the left and right versions modify the features of the same model element. However, if the changes from each version affect a

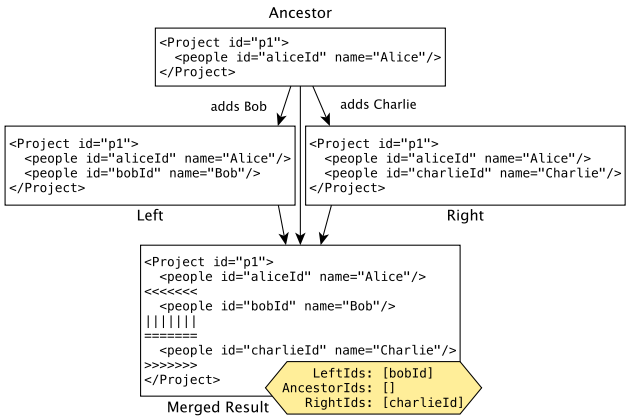

Fig. 8 Git marks as a conflict two additions of independent model elements that end up in the same model file line

different set of features, these changes could be automatically merged by combining the feature updates of each version. Figure 7 shows an example where the left version updates the start time of the *t1* task, while the right version updates its duration. In the merged model, the task is included into a conflict section, but an alternative merge is shown at the bottom right, where the updates to start time and duration are automatically merged and no conflict is reported. That is the default behaviour of PEACEMAKER: updates over disjoint sets of features are not considered a real conflict. Nevertheless, a defensive approach can be configured for those cases where enforcing a manual check is preferred, to determine if it makes sense to merge the changes coming from both versions.

Another way in which false positive conflicts can appear happens when two new and independent elements are added in the same line of a file, one on each model version. This case is depicted in Fig. 8, where two Person elements, *Bob* and *Charlie*, are added to the same project and end up serialised in the same line of each version. A conflict section would be generated by Git when merging this versions, and PEACEMAKER would identify both elements as Collateral Elements (described in the previous section). In some cases, this kind of conflict can be automatically processed by simply keeping both elements. However, if both new elements are added to the same containment feature (e.g. *Project.people* in Fig. 8), and that feature is defined as ordered, there remains the question of which new element should be placed before the other. As automatically selecting the order might not be correct in all cases, solutions such as EMF Compare resort to user input to determine which order to apply.

3.5 Conflict smells

Unfortunately, apart from the false positives described in the previous section, Git can also suffer from false negatives,

these are, cases where real conflicts exist, but the line-based merge operation misses them. In some cases, it is even worse, as the merge performed by Git introduces issues into the model that can even prevent it from loading. As some of these issues appear because of the existence of a real conflict, we can do the reverse: the presence of these issues in a merged model suggests a potential conflict missed by the merge operation. Consequently, we denote these issues as *conflict smells*. From the point of view of PEACEMAKER, conflict smells are treated as the conflicts of Sect. 3.3: these smells are reported, and resolution actions are offered. We describe the types of conflict smells in the following.

3.5.1 Duplicate ids

PEACEMAKER is able to find errors related to the presence of duplicated ids in the merged model. There are two possible sources of duplicated ids:

- Adding new elements in both versions of a model with the same identifiers. This can happen when element identifiers are obtained through an auto-incrementing counter, i.e. identifying tasks as *task1*, *task2*, and so on, and then adding a new *task3* in both left and right versions. Another potential gateway for this issue opens when an editable EAttribute is set as the id of model elements, so manual entries by end users can cause duplicate identifiers. This issue also includes adding a new element whose id matches the one of another element already present in the model.
- A model element is duplicated as a side effect of the line-based merge. This phenomenon can appear when model elements are moved to different positions in the left and right versions. For instance, Fig. 9 shows an example where a PSL Project contains four tasks, with ids ranging from *task1* to *task4* (as we just mentioned in the previous item, it is a bad idea to set ids based on incremental counters, but we do it in this example for clarity). In the ancestor version tasks are ordered by their ids in alphabetical order. However, *task1* is moved after *task3* in the left version, while it is moved to the end of the list in the right version, thus creating a conflict. When Git merges these model versions, and because of treating model elements as lines, it duplicates the *task1* task, which appears twice in the merged version: after *task3*, and at the end of the list. The same issue can also appear when an element is changed to a different container on the left and right versions. This case is depicted graphically in Fig. 10, where an Effort element is changed to two different tasks in left and right. When merging, the corresponding XMI lines of the moved element in its updated position on the left and right model versions are treated as new lines, so the element ends up appearing in two places in the merged model. Listing 3 shows the resulting merged model produced by Git, where the effort element has been duplicated in lines 3 and 6, and no conflict sections have been included. As Fig. 10 shows, this element was contained under the *originContainer* task (in line 8) in the ancestor version.

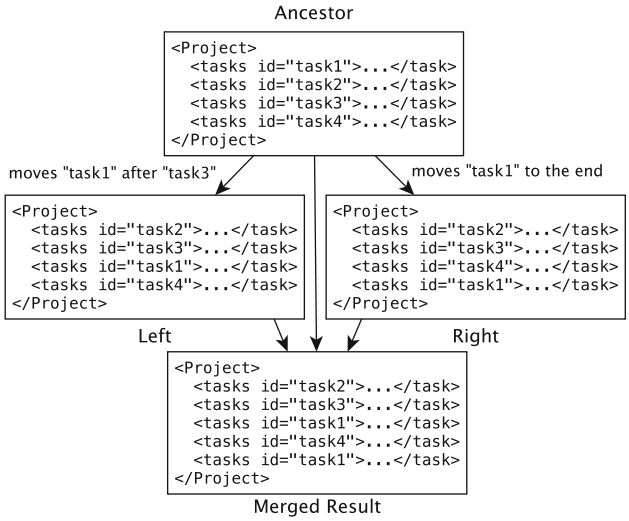

Fig. 9 Reordering of tasks in a PSL Project

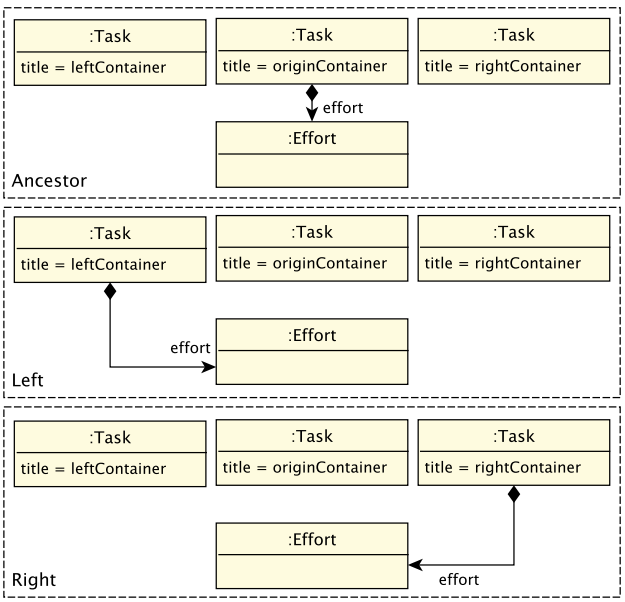

Fig. 10 Example where the *Effort* object changes to two different containers in left and right

When duplicated ids are found, PEACEMAKER offers resolution actions to decide which of the duplicates should be maintained.

```
1  <Project xmi:id="p1">
2    <tasks xmi:id="leftContainer">
3      <effort xmi:id="e1"/>
4    </tasks>
5    <tasks xmi:id="rightContainer">
6      <effort xmi:id="e1"/>
7    </tasks>
8    <tasks xmi:id="originContainer"/>
9  </Project>
```

Listing 3 The *e1* Effort element is duplicated as a result of merging the three model versions of Fig. 10

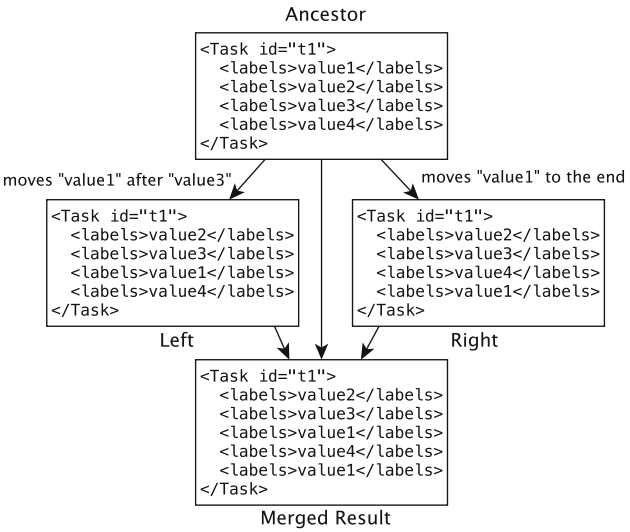

Fig. 11 Reordering the labels of task *t1* causes the duplication of *value1*

3.5.2 Duplicate values in multi-valued attributes

Multi-valued attributes are another place where line duplicates can cause errors. Figure 11 shows a task with several *labels* (label is a multi-valued-attribute in the PSL metamodel of Fig. 1). When Git merges this test case, it duplicates *value1*, which appears in the third and fifth positions of the *labels* attribute of the merged model. This case is equivalent to the one depicted in Fig. 9, where the duplication involved *Task* model elements. The duplication could be detected when validating the model if *labels* was defined to only contain unique values. Although this attribute is indeed defined as unique, for some multi-valued attributes repeated values could be allowed. As a protective measure, PEACEMAKER detects any duplication in a multi-valued attribute (unique or not) as a conflict smell, so users would need to check if the repeated values are correct, or if an unwanted duplication has happened, and thus duplicates should be removed. This leads us to recommend against the use of non-unique multi-valued attributes in domain-specific metamodels if line-based merging is to be used. We enumerate this and other considerations in Sect. 5.4.

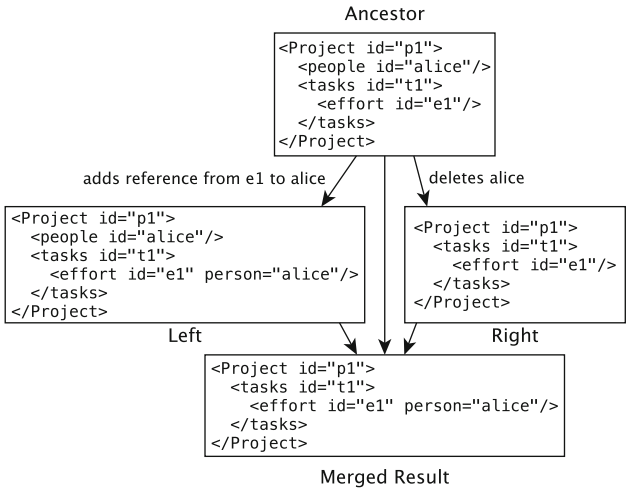

Fig. 12 A reference to a deleted element (*alice*) is introduced in the merged model

3.5.3 Internal dangling references

This conflict smell is caused by adding a reference from one model element to another in one of the versions and then deleting the referenced element in the other version. When versions are line-merged, the referenced element is indeed deleted, and the added reference is persisted. The combined effect of these two changes creates an internal (i.e. it does not involve external models) dangling reference in the merged model, which can cause an error when the model is loaded. Figure 12 shows this issue, where the addition of a reference from the *e1* effort to the *Alice* person on the left and the removal of *Alice* on the right introduces a dangling reference to a non-existing *Alice* element in the merged model. PEACEMAKER is able to detect this kind of dangling references, and they are shown along with other conflicts present in the model. Unfortunately, PEACEMAKER does not have enough information to provide other resolution actions apart from reporting the issue to end users, and then discarding the dangling reference. Any other solution would require a standard three-way comparison of the model versions.

3.6 Model saving and partial resolution of conflicts

Any conflict resolution action performed over the in-memory model versions must eventually be serialised back to disk to reflect the changes in the original model file with conflicts. To achieve this, PEACEMAKER first converts model versions to XMI individually in memory, and then these versions are merged in a line-based approach. One of the advantages of this approach is support for *partial resolution of conflicts*: any original conflict section can be serialised again when a user wants to save their changes over a model with conflicts, but not all conflicts have been resolved yet. To the best of

our knowledge, this is the first conflict resolution approach that allows saving a partially resolved model with remaining conflicts.

The three-way line-based approach to merge model versions works as follows: starting from the individually serialised versions, the lines of the left and right versions are compared sequentially. When lines match, they are serialised without any special change. As an analogy, matching lines would be given a *common* type according to the preprocessor introduced in Sect. 3.1. On the other hand, a left line and a right line that do not match indicate that a yet-unresolved conflict between the versions has been reached. To serialise this conflict, the structure of the original conflict section is recreated: first, the next matching line between the left, ancestor and right versions is calculated. Then, the set of lines from each version starting at the current line (i.e. the one that did not match) up to the calculated next matching line are serialised, starting with the left version, and followed by the ancestor and right versions. These line sets are surrounded with the appropriate `<<<<`, `||||`, `====`, and `>>>>` symbols to separate the conflict section segments.

3.7 Limitations

As mentioned at the beginning of Sect. 3, this approach requires that model elements have a unique identifier. A strong reason behind this requirement is the possibility of suffering from element duplication as a side effect of the line-based merging (described in Sect. 3.5.1). Similarity matching techniques [19] are usually applied in model comparison and merging when the elements have no identifiers. However, these techniques would be computationally very expensive to use to find element duplications, due to duplicates happening in single versions of the models. Therefore, every element of a model would need to be checked against all other elements of the same type, and this process needs to be carried out for each model version. More importantly, these techniques are approximate, and thus, they provide no guarantees on whether the resulting matched pairs are model duplicates or just false positives.

The approach has been devised and evaluated over EMF models, persisted using XMI, and versioned in Git repositories. In the following, we comment on any known limitations of applying this approach for other modelling and persistence technologies, or different versioning tools.

With respect to EMF, some conflicts described in Sect. 3.3 are explicitly related to EMF concepts, such as a Containing Feature Update for containment references; while others are not, like Double Update or Update Delete conflicts. A change to a different modelling technology could cause some conflicts not to apply, because of concepts such as containment not existing in the new technology. At the same time, any concept of the new technology that is not present in EMF might cause new kinds of conflicts to appear that could need special resolution. If these new conflicts are detectable and resolvable, we do not foresee a change in modelling technology to be a hard stopper to applying the described approach.

The model persistence format is of special importance, due to the low-level operation of the line-based merger. If the default XMI format were changed to a different one, such as JSON,[2] the first aspect to check is whether the line-based extraction depicted in Fig. 2 is able to provide the three model versions without formatting errors or inconsistencies, as it happens with XMI. If the versions format is correct, then the second requirement involves being able to gather the lines of the original file where each model element is persisted, so that conflict section identifiers can be populated (see Fig. 3). Lastly, a different persistent format could give room to new conflict false positives, apart from the ones currently covered in Sect. 3.4.

Lastly, using a different VCS other than Git should not represent a major issue, as the syntax to delimit conflict sections described in Sect. 2.3 is also used by other three-way comparison tools such as JGit[3] or GNU diff3,[4] or by other VCSs such as Mercurial[5] or SVN.[6] However, a thorough revision on whether the line-based merging algorithms of the alternatives are equivalent to the one provided by Git should be performed.

4 Implementation

This section presents the Eclipse-based software components we developed to implement the conflict detection approach described in the previous section. The source code of PEACEMAKER is available in a public repository.[7]

4.1 PEACEMAKER editor

For end users to visualise the detected conflicts and provide necessary inputs (i.e. resolution actions), PEACEMAKER provides an Eclipse-based editor that we describe here.

Figure 13 shows the contents of the PEACEMAKER editor when the model of Listing 2 is opened. On the left-hand side of this editor, there are four tree viewers. The left and right viewers at the top show the initial contents of the left and right versions of the model, respectively. The tree viewer

[2] https://emfjson.github.io/about/.

[3] https://www.eclipse.org/jgit/.

[4] https://www.gnu.org/software/diffutils/manual/html_node/diff3-Merging.html.

[5] https://www.mercurial-scm.org/wiki/TutorialConflict.

[6] https://svnbook.red-bean.com/en/1.7/svn.tour.cycle.html#svn.tour.cycle.resolve.diff.

[7] https://github.com/epsilonlabs/peacemaker.

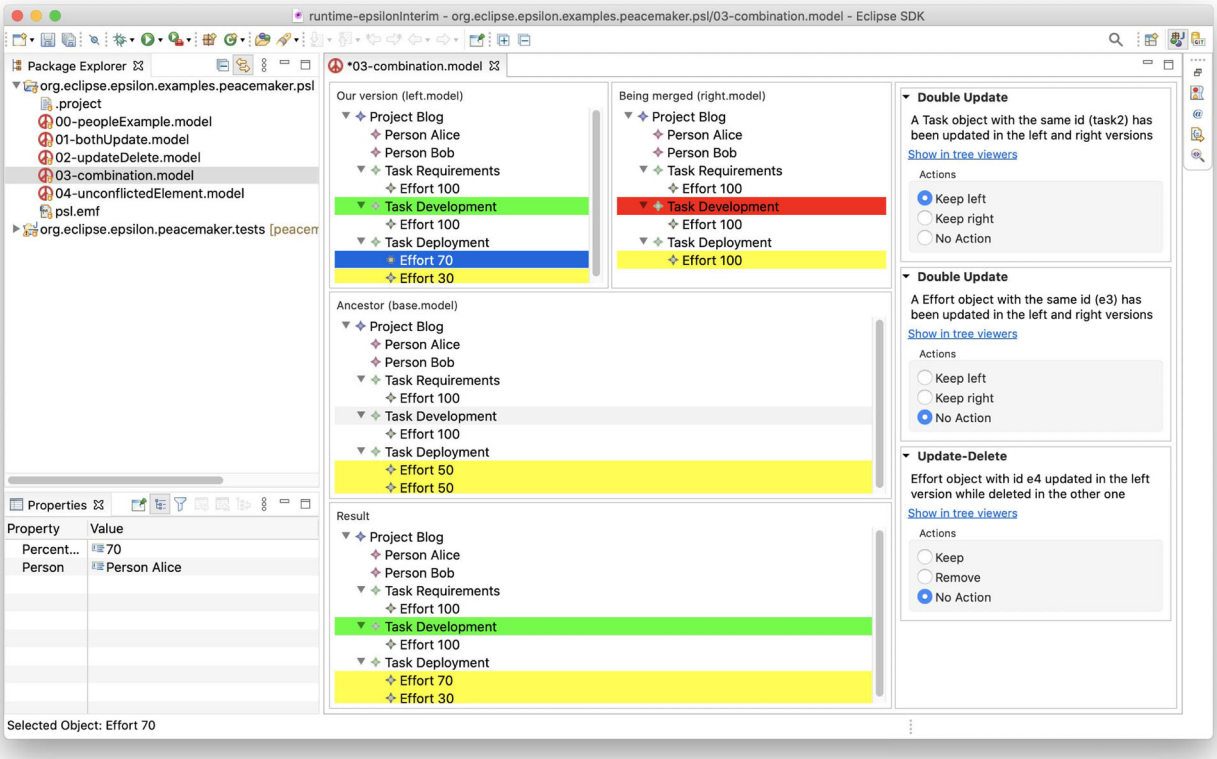

Fig. 13 PEACEMAKER editor showing the conflicts of the PSL model of Listing 2

in the middle shows the ancestor model version, with the objective of helping users to better detect the causes of the conflicts and to decide among the feasible resolution actions. Lastly, the result viewer appears at the bottom. While the left, right, and ancestor viewers always show the initial content as it was before performing any changes, the result viewer shows the updated content of the model after applying any resolution actions over the identified conflicts.

Resolution actions can be applied by interacting with the conflicts list, depicted on the right-hand side of the editor. This list enumerates the identified conflicts in the input model. For each conflict, it contains a brief description, a link that can be clicked to reveal and select the affected elements in the tree viewers, and a list of actions to apply represented as a set of radio buttons. When the user selects any of those buttons, the associated resolution action is invoked, and the content of the result viewer is updated to reflect the results of applying the selected action. PEACEMAKER also makes use of colours to indicate the state of conflicting elements. Green indicates an accepted element as a result of the resolution action. Red is used to mark rejected/removed elements, and yellow is used to highlight those elements taking part in yet-unresolved conflicts.

In the example of Fig. 13, the *Keep left* action has been selected to resolve the first conflict of the list. This conflict is of type Double Update (see Sect. 3.3), and it involves the *Development* task. As the left version has been selected, the corresponding task element on the top left tree viewer is highlighted in green, while the discarded one on the right version appears in red. The *Development* task in the result viewer at the bottom is also highlighted in green to reflect that the conflict regarding that element has been resolved. The two remaining conflicts relate to the person efforts of the *Deployment* task. As these conflicts have not been resolved yet, the affected elements are highlighted in yellow in the tree viewers.

Standard editor operations such as undo and redo are supported. Also, PEACEMAKER supports other features typically found in model editors. For instance, we can inspect the contents of model elements from any of the tree viewers by using the Properties view that can be seen at the bottom left of Fig. 13. In the figure, this view shows the properties of the Effort element selected in the top left tree viewer (highlighted in blue).

After applying any of the available resolution actions, the model can be saved to persist the changes. If not all existing conflicts have been resolved yet, then a partial save is per-

formed, as described in Sect. 3.6. This way, a user can save their work at any point while resolving a conflicting merge operation, just as it is usually possible when working with conventional text editors and source code files.

4.2 Using a custom merge strategy

VCSs allow replacing the default line-based merge operation with custom merge strategies for certain selected types of files. In the context of models, a custom strategy can, for instance, help avoid false positives, or prevent missing some conflicts.

The EGit project,[8] which provides Git support to the Eclipse IDE, allows defining alternative merge strategies. For instance, EMF Compare [11] provides its own merge strategy that performs the standard full three-way model comparison when merging model files to detect any conflicts. We have implemented a custom strategy that uses PEACEMAKER to improve the Git line-based merge. Precisely, this strategy takes the following steps:

1. It starts by merging the versions with the default line-based merge. If lines are found to be in conflict, this step includes conflict sections in the model.
2. The merged model is loaded with PEACEMAKER following the steps of Sect. 3. This load removes any false positives of the Git merge (see Sect. 3.4). In some cases, it might be determined that the conflict sections only contained false positives. In addition, the PEACEMAKER load also detects conflict smells that can be a hint for Git potentially missing some conflicts, e.g. duplicated ids or dangling references (see Sect. 3.5).
3. If real conflicts or conflict smells are found by PEACEMAKER, the model file is marked as in conflict.

So, this merge strategy only requires to load the model once, instead of performing a full three-way model comparison as solutions like EMF Compare do. After a Git merge, we could also just open the merged model in the PEACEMAKER editor to perform the same checks, but configuring a merge strategy to do that automatically after every merge can be more convenient, e.g. it saves users time when several models have been merged, and it prevents from users forgetting to open some of the merged models.

5 Evaluation

For the evaluation of our approach, we focused on two aspects. The first aspect assesses the completeness of PEACEMAKER, by determining if it is able to detect the different types of conflicts that can appear when merging model versions. Then, the second one involves a quantitative evaluation of the performance and scalability of PEACEMAKER against two state-of-the-art approaches.

5.1 Completeness analysis

For the first part of this evaluation, we studied the ability of our approach to detect a variety of conflicts in models. For that purpose, we tested PEACEMAKER with an external suite that contains a comprehensive set of the types of conflicts that can appear when merging EMF model versions. This suite is part of the unit tests used internally by EMF Compare[9] (introduced in Sect. 2.4), ensuring that all versions of this tool are able to detect the conflict types of the suite.

Table 2 summarises the tests contained in the conflicts suite. Tests are organised in 11 categories, from *a* to *k*. Each category exemplifies a type of conflict that might arise in XMI-based models. As an example, *a* refers to tests updating the value of a single-valued feature of a model element in one version, while deleting the same element in the other (i.e. an *UpdateDelete* conflict as described in Sect. 3.3); another category, *c*, contains similar changes, but the affected features are multi-valued. Additionally, there are several test cases for some categories, representing different ways in which conflicts of the same type might appear. For instance, a_1 updates a feature with a new value on the left, while deleting the containing element on the right. Similarly, a_3 deletes the element on the right, but to update the feature on the left it unsets its value, instead of providing a new one. Categories *a* to *d* are expanded into the different types of model features where the conflict might appear, namely, EAttributes and non-containment EReferences (*attr* and *ref* in the table). As for containment EReferences (*contRef*), the test suite included a test for the *b* category (i.e. single-valued features), and we extended the suite by adding tests for multi-valued containment references in the *d* category. Lastly, while most cases contain conflicts, there are some negative cases where no conflicts are present. Thirteen test cases, namely b_5–b_6 and d_4–d_6, apply the same changes to both sides, so no conflicts should be detected for these cases. We describe the meaning of the table cell values in the next sections, along with a discussion of the obtained results.

5.1.1 Issues with line-based merging of models

The original EMF Compare suite provides the three model versions of each test case, these are, the left, ancestor and right versions. So, the first step to use this suite with PEACEMAKER

[8] https://www.eclipse.org/egit/.

[9] https://git.eclipse.org/c/emfcompare/org.eclipse.emf.compare.git/tree/plugins/org.eclipse.emf.compare.tests/src/org/eclipse/emf/compare/tests/conflict.

Table 2 EMF Compare conflicts test suite, organised by modification categories, feature type (if relevant) and case number

Category	Description	Feature	T_1	T_2	T_3	T_4	T_5	T_6
a	Left changes the value of single-valued feature. Right deletes the containing element	attr	+	+	+			
		ref	✓	✓	✓			
b	Left changes the value of a single-valued feature. Right changes the same feature to a different value	attr	✓	✓	✓	✓	✓	✓
		ref	✓	✓	✓	✓	✓	✓
		contRef			✓			
c	Left changes the value of a multi-valued feature. Right deletes the containing element	attr	+	+	+	+	+	
		ref	✓	✓	✓	✓	✓	
d	Left changes a multi-valued feature. Right changes the same feature in an incompatible way	attr	!	!	∼	✓	✓	✓
		ref	✓	✓	✓	✓	✓	✓
		contRef	!	!	∼	✓	✓	✓
e	Adds reference towards deleted object	-	∼	∼	∼			
f	Changes containment feature of object to two distinct features in left and right	-	✓					
g	Changes container of object to two distinct objects in left and right	-	∼					
h	Changes container of object to a deleted one	-	!	!				
i	Deletes different sets of elements in left and right	-	+					
j	A combined modification of features with deletion of contained elements	-	+					
k	Adds objects to left and right (sometimes with the same identifiers)	-	∼	∼	∼	✓		

Symbols indicate if the Git merge operation identified the case correctly (✓); whether Git wrongly marked false positives (+); whether PEACEMAKER was able to detect conflict smells after a wrong Git identification (∼); or those cases with conflicts that were missed by Git and PEACEMAKER (!)

involved the generation of merged models with conflict sections such as the one of Listing 2, which we achieved by using the *git merge-file* command.[10]

As we have seen in Sect. 3, the merge operations performed by Git are line-based, so it is irrelevant whether the merged lines represent source code such as Java or C, or if they contain (semi-)structured data like JSON or XML. While this is not an issue for the majority of test cases, we found that some of them are not properly handled by Git's merge operation, including concerning ones where conflicts are not detected and the merge operation introduces errors in the model. We consider as an error anything that breaks the well-formedness of the model, such as references that point to non-existing elements, or a violation of any conformance rules of the metamodel (i.e. feature cardinalities, uniqueness). We also consider as errors unwanted duplications of model elements or values that cannot be automatically detected and fixed.

We categorised the results of Table 2 into four groups, based on the outcome of Git's merge operation and the posterior analysis by PEACEMAKER. The test suite is composed of 60 cases, in which there are 47 cases that contain conflicts, and 13 cases that do not. As all 13 cases without conflicts were correctly identified by Git, the separation in groups focuses on categorising those cases that contain conflicts, as the results for these were not homogeneous.

Figure 14 shows a flowchart with the possible paths that a test case with conflicts could follow until it was assigned into one of the four result groups. The flowchart starts from the 47 test cases that contain real conflicts of some kind. The first step involves the line-based merge performed by Git. Cases marked with a check (✓) indicate that the presence of conflicts was correctly detected by Git, and then by PEACEMAKER. When Git identification was incorrect, it was due to either false positives or false negatives. Git false positives, marked with a plus symbol (+), were correctly detected and omitted by PEACEMAKER. As for Git false negatives, PEACEMAKER was able to detect conflict smells (see Sect. 3.5) in almost all of them by looking for issues such as duplicated ids, internal dangling references, or broken conformance rules (e.g. unique multi-valued attributes with duplicates). We marked these cases with a tilde (∼) symbol. Lastly, there were a subset of cases missed both by Git and PEACEMAKER, which we marked with an exclamation (!). Although missed, Git did not introduce any structural errors in the model. The following sections give more details about these four groups.

[10] https://git-scm.com/docs/git-merge-file.

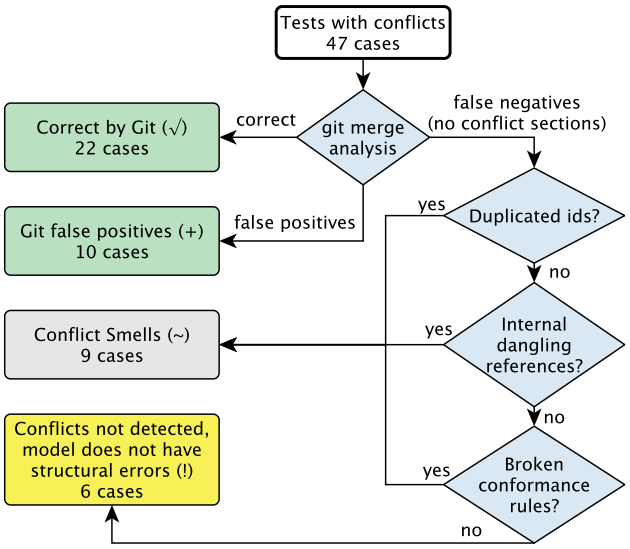

Fig. 14 Flowchart that explains the different possible outcomes of the test cases with conflicts of Table 2

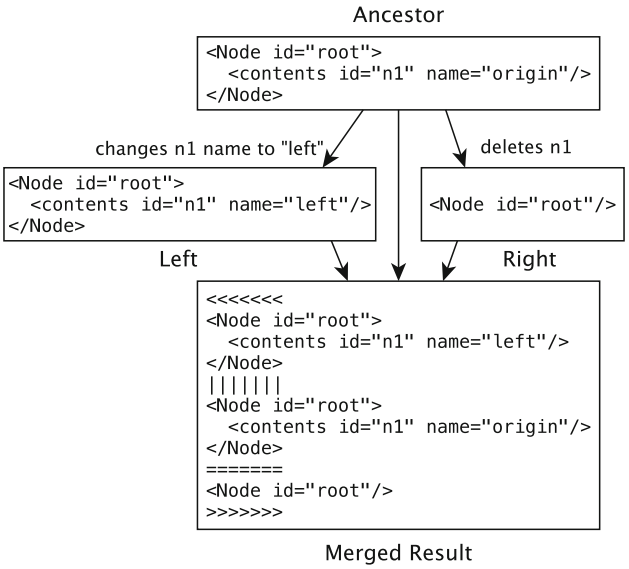

Fig. 15 Example of a Git false positive ($a_{1_{attr}}$): the *root* element is wrongly identified as in conflict because of a change in how its tag is ended on the right (/> instead of >)

5.1.2 Correct git identification (✓)

These tests cases are correctly processed by the Git merge. In all tests where this happens, PEACEMAKER is also able to identify the underlying conflicts in the models. As we mentioned, the 13 cases that contained no conflicts were properly identified, so they also belong to this group. This was somewhat expected, as after applying equivalent changes to the left and right model versions the persisted XMI lines of the affected model elements were equal. With 35 out of the 53 test cases correctly identified, this is the most frequent outcome of the test suite.

5.1.3 Git false positives (+)

There are 10 cases where Git manifested false positive issues. First, in 8 positive test cases (i.e. with real conflicts), Git marked additional model elements as in conflict because of a change in the closing tag of an element whose contents have been deleted. This type of false positive was described in Sect. 3.4. As an example, Fig. 15 depicts the ($a_{1_{attr}}$) case, where the *root* element is wrongly included in the conflict section because of a change in its starting tag from > to /> in the right version.

Second, we include here two special cases that are considered false positives in the EMF Compare test suite, as they do not represent real conflicts at the model level. These conflicts belong to the *i* and *j* categories and involve deletions of different sets of elements on the left and right versions. Figure 16 depicts the *i* category case. We can see that, on the right version, only the *A3* model element contained in *C* is deleted, while on the left the *C* element is deleted instead,

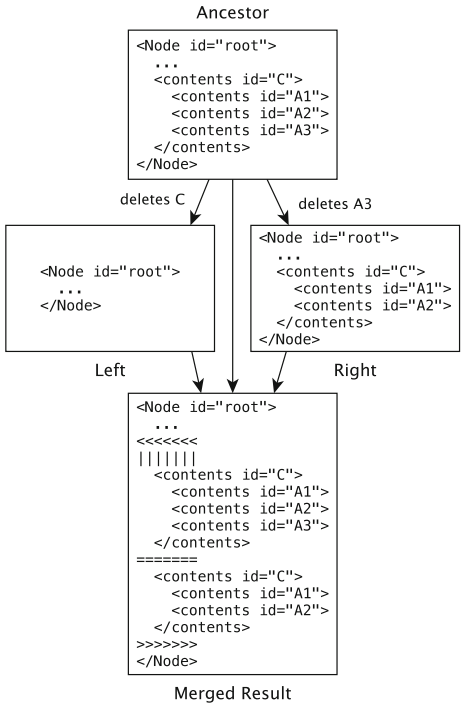

Fig. 16 Git identifies as a conflict two deletions affecting distinct sets of elements (*i* category test case)

which also removes all its contents (*A1*, *A2* and *A3*). Here, EMF Compare automatically assumes that the *A3* deletion is a subset operation of the deletion of *C*, which indirectly also removes *A3*. While we could do the same in PEACEMAKER, we included the KeepDelete conflict to mark cases where an element is deleted in one version, but is kept unmodified in the other (see Sect. 3.3). Our rationale for this is that the

difference in the set of deleted elements between the versions could be of interest for the end users, so we leave the final decision on which elements to keep or remove to them. Nevertheless, the decision on whether to treat this case as a conflict that requires manual intervention or to automatically perform a merge could be easily managed as a configuration parameter of the PEACEMAKER identification process.

5.1.4 Git false negatives with conflict smells (∼)

This group contains those cases where Git does not detect any conflicts, but PEACEMAKER is able to find conflict smells that, as we presented in Sect. 3.5, are often caused by unwanted errors after a line merge. There are two options to detect these smells. First, models can be opened with the PEACEMAKER editor after performing a merge with Git, even in those cases where no line conflicts have been detected. Alternatively, we presented a custom merge strategy to automatically perform this check after each Git merge in Sect. 4.2. The identified conflict smells for the EMF Compare test suite are detailed in the following.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Node ...>
3    <contents
4        xmi:id="_-iLZEJReEeGwLqrAWz-_6w"
5        name="leftContainer">
6      <contents
7          xmi:id="_CRQ58JRfEeGwLqrAWz-_6w"
8          name="conflictNode"/>
9    </contents>
10   <contents
11       xmi:id="__cxZEJReEeGwLqrAWz-_6w"
12       name="rightContainer">
13     <contents
14         xmi:id="_CRQ58JRfEeGwLqrAWz-_6w"
15         name="conflictNode"/>
16   </contents>
17   <contents
18       xmi:id="_AY8tcJRfEeGwLqrAWz-_6w"
19       name="originContainer"/>
20 </Node>
```

Listing 4 Test case from the *g* category where the *conflictNode* element has been duplicated by the line-based Git merge (lines 6-8 and 13-15)

Duplicated Ids

This type of smell, which was introduced in Sect. 3.5.1, appears in several cases of the test suite. Conflicts k_1 to k_3 add new elements in both versions of the model with the same identifiers. Duplications also appear in the $d3_{contRef}$ case due to incompatible reorderings of model elements, which is equivalent to the duplicated tasks example of Fig. 9. Listing 4 depicts an additional case of element duplication from the *g* category, where the container of the *conflictNode* element is changed to two different containers in each version: *leftContainer* in the left, and *rightContainer* in the right. The initial container of this element in the ancestor version is *originContainer*. As a result of this incompatible change of containers, the *conflictNode* appears twice in the merged model: first in lines 6-8 under *leftContainer* and second in lines 13–15 under *rightContainer*.

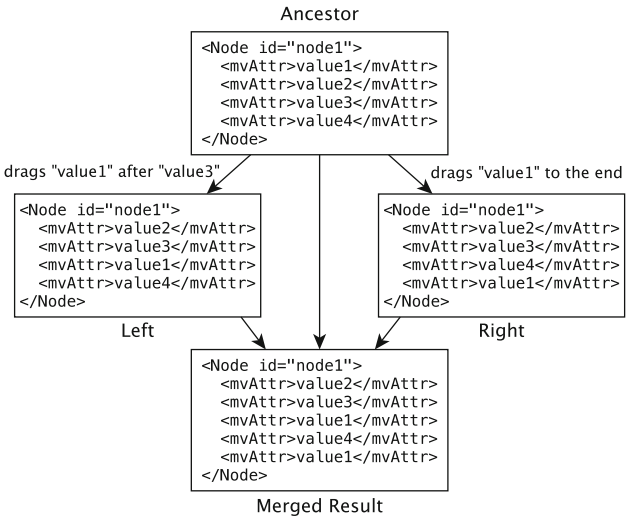

Fig. 17 $d3_{attr}$ test case with conflicting reorderings to a multi-valued attribute (*mvAttr*)

Duplicates in Multi-Valued Attributes

We also found duplications in multi-valued attributes. Figure 17 depicts the ($d3_{attr}$) case, which is identical to the duplication described in Fig. 11 over PSL task labels. When Git merges this test case, an unwanted duplication of one of the values (*value1*) takes place.

Internal Dangling References

The *e* category contains conflicts related to the creation of internal dangling references, which were described in Sect. 3.5.3. Figure 18 shows the *e1* case, where a reference from *conflictHolder* to *origin* is added on the left, while *origin* is deleted on the right, thus creating a dangling reference to a non-existing *origin* element.

5.1.5 Conflicts not detected, model has no structural errors (!)

This group contains the test cases where PEACEMAKER does not have enough information to detect the conflicts after Git missing them. In this group, the line-based merge does not introduce any structural or duplication-related errors (as defined in Sect. 5.1.1) in the merged model.

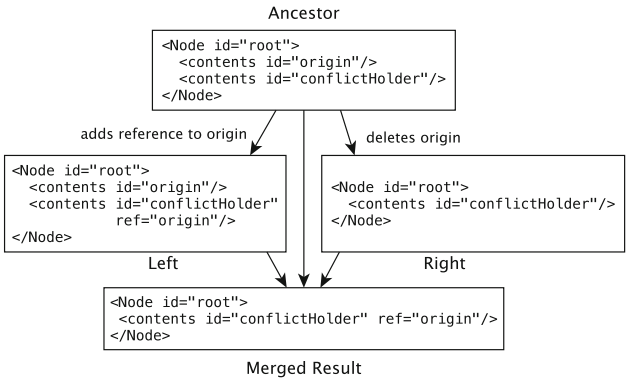

Fig. 18 e_1 test case where a reference to a deleted element (*origin*) is introduced in the merged model

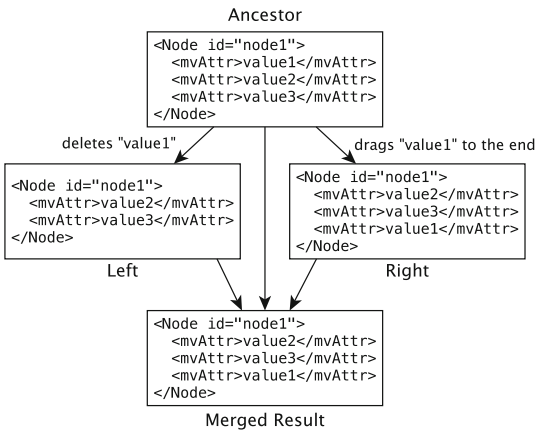

Fig. 19 $d_{1_{attr}}$ test case with conflicting left and right modifications to a multi-valued attribute (*mvAttr*)

Figure 19 depicts a simplified test case ($d_{1_{attr}}$) belonging to this group. In the example, we have a *node* with a multi-valued attribute denoted as *mvAttr*. In the ancestor version, *mvAttr* contains three values: *value1*, *value2* and *value3*. The conflict is caused by the modifications over *value1*: while in the left version that value is deleted, in the right one it is moved to the latest position of the multi-valued attribute. When these versions are merged, the original line belonging to *value1* disappears in both versions (i.e. no *value1* above *value2*), while the right *value1* appearing after *value3* is merged as a new line. As a result, *node.mvAttr* ends up containing *value2*, *value3* and *value1* in the merged model. While the result matches with one of the initial versions (i.e. the right one), the way in which this case was merged was beyond the control of the developer. This case is equivalent to $d_{1_{contRef}}$ where, instead of multi-valued attributes, the modifications take place over a multi-valued containment reference. Lastly, $d_{2_{attr}}$ and $d_{2_{contRef}}$ are mirrors of the respective first two cases, this is, left changes are applied on the right version, and vice versa.

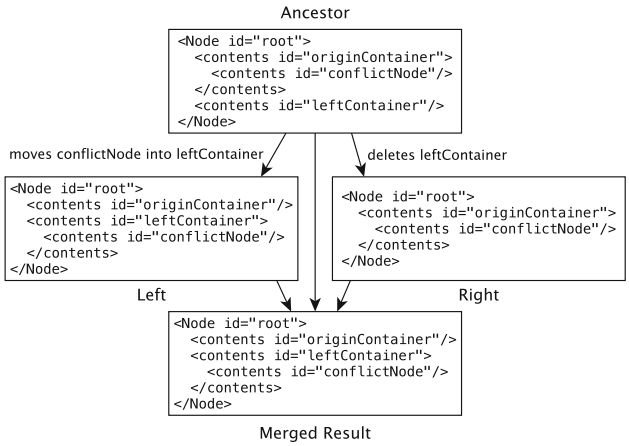

Fig. 20 h_2 test case where the container to the *conflictNode* is changed to *leftContainer* on the left version, while that same *leftContainer* element is deleted on the right version

The *h* cases, also belonging to this group, present a similar situation in which no conflict is detected and the VCS automatically decides how to combine the changes coming from the two versions. This category of conflicts involves changing the container of an element to a new parent in one of the versions, while deleting the new parent in the other version. Figure 20 depicts the h_2 case, where the container of the *conflictedNode* element is changed to *leftContainer* on the left, while on the right version that new container is deleted. As shown at the bottom of the figure, the line-based merge automatically selects the left version of the changes and disregards the deletion taking place on the right version.

We summarise the results of the completeness analysis in Sect. 5.3 along with the results of the performance comparison, which comes next.

5.2 Performance comparison

We measured the time required by PEACEMAKER and two state-of-the-art tools to detect and identify conflicts in contrived models of increasing size. These measurements were taken using the Java Microbenchmark Harness (JMH) tool,[11] and details about the parameters used to configure the benchmarks and the computing platform can be seen in Table 3. Also, the benchmarking code is openly available in an external repository[12] for reproducibility purposes. The next sections describe the experiments carried out in more detail.

5.2.1 Compared approaches

We compared PEACEMAKER against EMF Compare [11] and EMF DiffMerge [12], which were introduced in Sect. 2.4.

[11] https://openjdk.java.net/projects/code-tools/jmh/.

[12] https://github.com/alfonsodelavega/peacemaker-evaluation.

Table 3 Computing platform and benchmark parameters used during the performance comparison

Computing Platform	
Type	Laptop (MacBook Pro)
CPU	Intel 4-core/8-thread
RAM	32 GiB
Storage	NVME SSD Drive
Benchmark Parameters	
Warm-up Iterations	5
Measured Iterations	10
Error Markers	99.9% Confidence Intervals
JMH Mode	Average Time (2s/iteration)
XMI Load Options	`Defer_IdRef_Resolution`

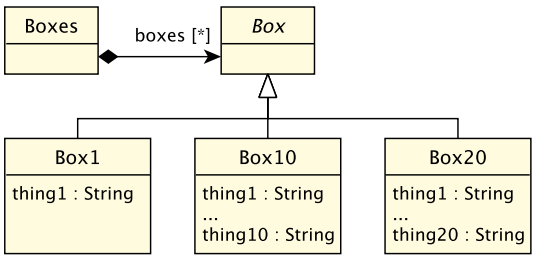

Fig. 21 Boxes language metamodel in Ecore

Both tools perform a three-way model comparison to find differences and conflicts, so they are ideal candidates to determine if the line-based approach used in PEACEMAKER provides any performance improvements. As for PEACEMAKER, we included two variants: a sequential one, where the split model versions (see Sect. 3.2) are loaded one after the other, and a parallel variant that uses multi-threading to load all versions at the same time. Lastly, we also included the standard XMI load time of the left model version as a baseline that could help extrapolate the results to other computing platforms.

5.2.2 Types of models

We generated models conforming to two different metamodels. The first metamodel is the Project Scheduling Language used as running example throughout the paper and introduced in Sect. 2.1. For the second one, we defined a Boxes language whose metamodel is depicted in Fig. 21. This is a very simple language that can be used to define a list of *Boxes* of varying size, this is, with a different number of attributes or *things*. For instance, *Box1* instances have 1 attribute, while *Box10* and *Box20* have 10 and 20 attributes, respectively. Boxes models were useful to test whether having denser model elements (i.e. with more attributes to check) had any effect on the completion times of the compared approaches

5.2.3 Comparison scenarios

The three compared approaches were used to detect conflicts in several comparison scenarios. Each scenario is composed of different test cases of increasing size, from 1000 to 200 K elements. This size refers to the number of tasks in PSL models, and to the number of boxes in Boxes models. For each scenario and test case size, we generated ancestor, left, and right model versions, as these are required for EMF Compare and DiffMerge. To obtain a merged input model for PEACEMAKER, we used the same *git merge-file* command as in the completeness evaluation. We included the time it took this command to create each merged model as part of the conflict detection times for both PEACEMAKER variants.

Given a test case with a size of N elements, the ancestor version was generated as follows:

- PSL models with N tasks and 5 people. Each task effort is shared 50–50 between 2 people (chosen at random, with fixed seed for reproducibility).
- Boxes models with N boxes.

Over this ancestor, conflicting left and right versions were created. Unless stated otherwise, the number of conflicts that was included in a test case was fixed to 10. The following conflict scenarios were generated:

1. *PSL Update Delete*: in conflicting tasks, the percentage effort of one person was updated on the left, while the whole effort element was deleted on the right.
2. *PSL Double Update*: we updated the title of conflicting tasks to different values on left and right. The objective of this scenario is to try a different kind of conflict. Also, PEACEMAKER performs false positive checks for Double Update conflicts (see Sect. 3.4), so we wanted to check if that affects the detection times.
3. *PSL Update Delete with extra changes/conflicts*: this is an extension of the first scenario where, apart from conflicts, we also introduced extra changes that did not create conflicts, with the objective to see how the compared approaches responded to them. Section 2 introduced the comparison process for both EMF Compare and EMF DiffMerge, which compare the model versions of the conflicting branches This process starts by finding all differences between the two versions and then determining if any of these differences are conflicting. We included changes to 10, 50 and 100% of the tasks, excluding the ones with conflicts. Similarly, we also tested increasing the number of conflicts in the test cases, to see if there was any difference with including non-conflicting changes. The number of conflicts was again 10, 50 and 100% of the model tasks for each size. While the cases involving 50% and 100% changes/conflicts are unrealistic in

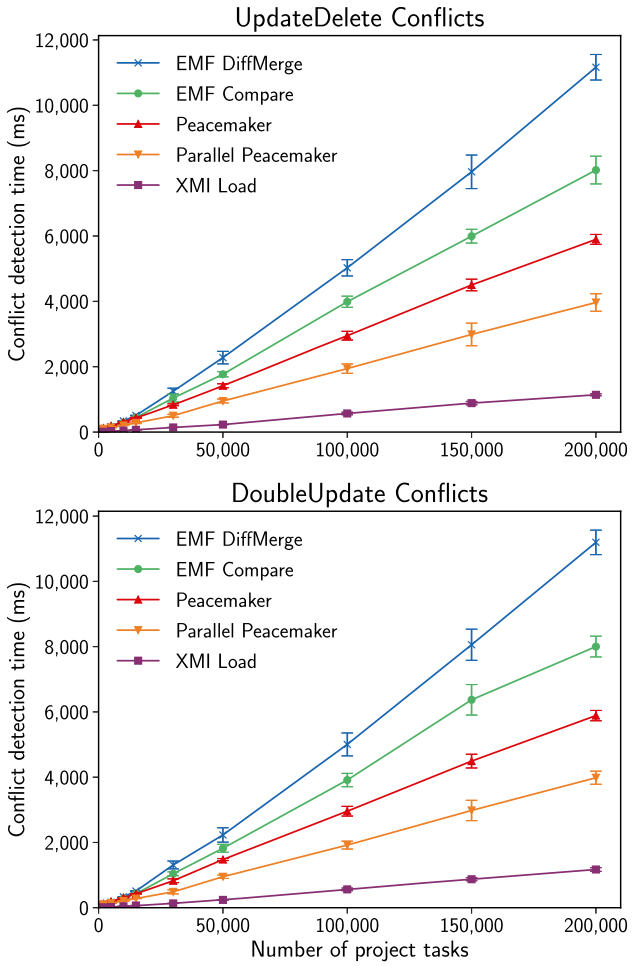

Fig. 22 Detection times of scenarios 1 (top) and 2 (bottom)

practice, we included them as stress tests to see how the compared approaches behaved in extreme scenarios.

4. *Boxes Update Delete*: we modified the first attribute of conflicting boxes in the left and deleted these boxes in the right. This scenario was repeated for the three box sizes (1, 10 and 20 attributes per box instance), to see if having bigger model elements had an impact on the times of the compared approaches.

5.2.4 Results

Figure 22 shows the conflict detection times for scenarios 1 and 2. Starting with the first scenario (top chart), we can see that both the sequential and parallel versions of PEACEMAKER are faster than EMF Compare and DiffMerge. This phenomenon is consistent for all subsequent scenarios. While the time differences are not very critical for the smaller cases (i.e. identification times are below 1–2 s), for models with 50 K tasks and above these differences start becoming noticeable. The relative distance between the obtained times remains consistent when increasing the size of the models. If we average the differences for the 50 K, 100 K, 150 K and 200 K tasks models, then the sequential PEACEMAKER variant required 42% less time than EMF DiffMerge, and 24% less time than EMF Compare. In the case of the parallel variant, the reductions increase to 61% and 50%, respectively. When comparing the two PEACEMAKER variants, performing a parallel load provides a 33% reduction on conflict detection times, which makes it a third faster than the sequential variant. Lastly, and focusing again on the average results for the four largest models, EMF DiffMerge took 9.38 times the duration of the XMI load of a single model version, while this value was 7.13 for EMF Compare, 5.40 for the sequential PEACEMAKER, and 3.60 for the parallel PEACEMAKER.

Focusing now on the second scenario (Fig. 22, bottom), the times for all the approaches are very close to the ones of the first scenario. The change in the type of conflict found in the models, from Update Delete conflicts in scenario 1 to Double Update conflicts in scenario 2, caused less than 2% variation in times for EMF Compare and DiffMerge on average, and less than 1% for the PEACEMAKER variants.

The results for scenario 3 are depicted in Fig. 23. The top row shows the results when conflicting models have 10% extra non-conflicting changes, on the left, and two more extreme cases of having 50% and 100% non-conflicting changes in the middle and on the right. When compared with the results of scenario 1, EMF Compare took 2%, 9% and 17% longer times to detect the conflicts for the models with 10%, 50% and 100% extra non-conflicting changes, respectively. EMF DiffMerge suffered a bigger slowdown, taking 4%, 19% and 35% longer to complete the task. Nonetheless, despite the greater penalty for the cases with a large number of differences, the results suggest that a realistic amount of non-conflicting changes would not pose an issue to these two approaches. As for the PEACEMAKER variants, there was a small increase in the detection times, as a result of the extra time taken by the line-based merge command to process the non-conflicting changes. The increases for the sequential and parallel PEACEMAKER variant with respect to the scenario 1 results were of less than 3% in all cases.

The bottom row of Fig. 23 shows what happens when the number of conflicts in a model grows, starting with 10% conflicting tasks on the bottom left (i.e. in a model with 1000 tasks, 100 of them would have conflicting changes), up to the extreme cases with 50% and 100% conflicting tasks in the bottom middle and right, respectively. While these increases did not noticeably impact EMF DiffMerge results (1%, 2% and 6% longer times for the models with 10%, 50% and 100% conflicts), they did affect PEACEMAKER variants (5%, 13% and 22% longer times for the sequential variant, and 4%, 18% and 33% for the parallel variant), and EMF Compare (4%, 14% and 25% longer times). Again, these results show that the approaches behaved decently against extreme

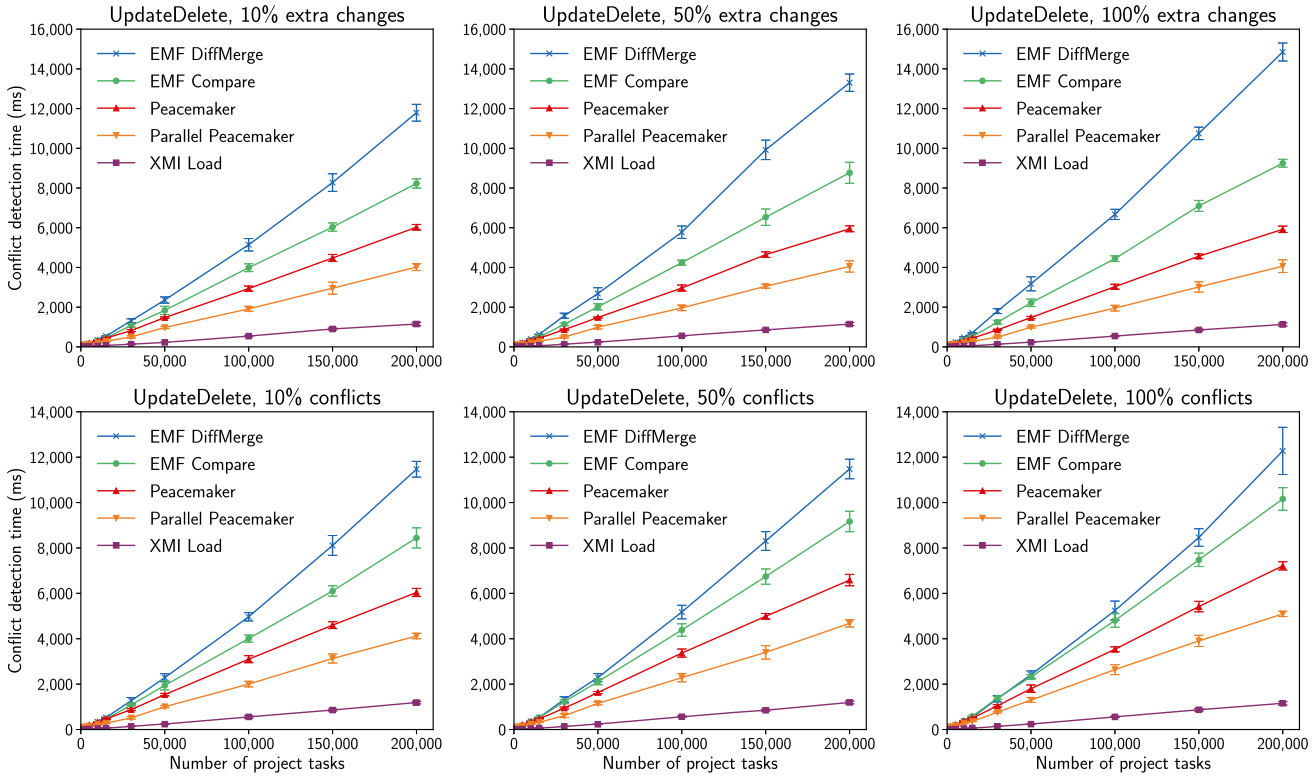

Fig. 23 Detection times of scenario 3. The top row shows the cases with 10%, 50% and 100% non-conflicting extra changes, and the bottom row shows cases with 10%, 50% and 100% conflicting tasks

conflict numbers, which indicates that there should not be any performance issue when applying these approaches to models with more realistic (i.e. lower) conflict numbers.

Finally, Fig. 24 covers the detection times for the Boxes models. In general, the conflict identification times in these models were lower than for the PSL ones, e.g. the slowest approach took up to ∼ 4 s for the Boxes model with Box1 instances (Fig. 24, left), while it took up to ∼ 10 s for the PSL model of scenario 1 (Fig. 22, top). We attribute this difference to the existence of more model elements in PSL models, as each Task instance also contains two Effort objects. In addition, the loading process of PSL models has to resolve the cross-reference that each task effort has to the person assigned for its completion, while in the Boxes models there are no references between the model instances.

Focusing on the results of the compared approaches, we can see that there is a general increase in the detection times of models with bigger Box elements, i.e. the times for Box10 instances in the middle chart are bigger than those of Box1 instances on the left chart; and the longest times are seen for Box20 elements on the right. This is partly caused by having bigger XMI load times, which took 0.29 s to load the largest Box1 model (200 K instances), while it took 0.76 and 1.33 s to load models with the same number of elements but larger Box10 and Box20 instances, respectively. As for the result of the different approaches, EMF DiffMerge results were 1.56 and 2.28 times slower for the models Box10 and Box20 instances, with respect to the ones with Box1 instances. Performing analogous comparisons, EMF Compare was 2.03 and 3.33 times slower, sequential PEACEMAKER was 2.04 and 3.3 times slower, and the parallel variant of PEACEMAKER was 1.91 and 2.96 times slower for Box10 and Box20 instances than for Box1 ones.

We can see that sequential and parallel PEACEMAKER kept being the fastest approaches across all Boxes models versions, although in this case the times for the sequential PEACEMAKER were closer to those of EMF Compare than in the previous cases. This closeness might be partly explained because of an existing issue that we experienced while using EMF Compare with our Boxes models. This issue caused an out-of-memory heap error (still present when using up to 25 GiB of heap size) when checking the ordering of containment references with more than 100 K values. Therefore, to be able to complete these tests, we disabled ordering checks with a custom EMF Compare differencer, which means that this approach performs less work than in the test cases with PSL models. We also found that the issue had already been registered as a bug in the past, so we contributed an example project with a comparison that allows reproducing it.[13]

[13] https://bugs.eclipse.org/bugs/show_bug.cgi?id=432497.

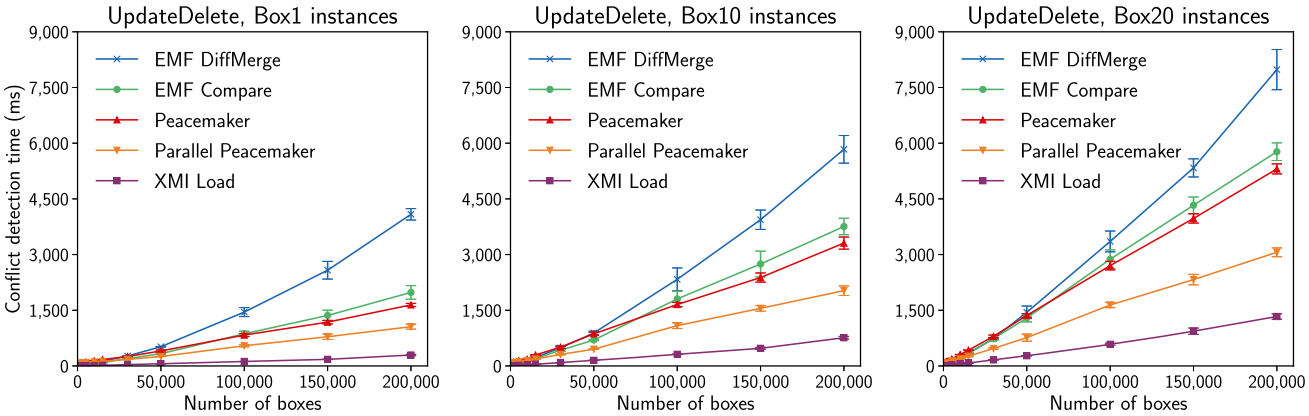

Fig. 24 Detection times of scenario 4, for each type of box instance

Focusing on the denser Box20 instances models (Fig. 24, right), sequential PEACEMAKER needed 21% and 4% less time on average than EMF DiffMerge and EMF Compare for the largest 4 models. The parallel PEACEMAKER variant was able to perform better, achieving a time reduction of 54% and 44% against EMF DiffMerge and EMF Compare.

5.2.5 Threats to validity

The main treat to the validity of the presented results is that the measured performance could be specific to the generated models for the tests, to the type of model, or to the included conflicts. Although performing the evaluation using real, third-party models would have been the ideal scenario, shortage of large publicly available real-world models is a widely recognised issue in model-driven engineering research (e.g. [21]). Our motivation for PEACEMAKER comes from our work with industrial partners [22] where we routinely encounter models containing many thousands of model elements, for which tools such as EMF Compare can take several seconds to perform a full 3-way comparison. As we are not able to meaningfully report on experiments using such models due to intellectual property reasons, we have opted for large synthetic models conforming to a metamodel that provides a comprehensive coverage of the features of Ecore (e.g. containment/non-containment references, single/multi-valued and ordered/unordered features). Generating models conforming to one or other metamodel is not immediately relevant, as the compared approaches search for conflicts in structural changes, and do not analyse semantics. The only reason behind having two different metamodels in our tests was testing model elements with different number of attributes (i.e. a structural difference), to assess if that affected the conflict detection times for any of the compared approaches. With respect to the created conflicts, we focused our performance evaluation around conflicts that could be detected by all the applied tools. We covered the detection of corner case conflicts in our completeness analysis of Sect. 5.1.

The longer identification times of EMF Compare and DiffMerge could be due to the fine-grained analysis they perform to detect changes in model elements' features, and it could be argued that if PEACEMAKER performed this fine-grained analysis, its times would be equivalent to the ones of the other approaches. This is not the case, as conflicting lines identified by the VCS already tell PEACEMAKER which model elements contain the conflicts without the need to actually check the features of all elements in the model. If fine-grained analysis of the model element features were needed in PEACEMAKER, it would only require processing this set of pre-identified conflicting elements, instead of the full model. Moreover, PEACEMAKER already performs this fine-grained comparison of element features when searching for false positives of the line-based merge (see Sect. 3.4), so the PEACEMAKER detection times shown in the performance comparison already include checks at the feature level of model elements.

Finally, it could be argued that parallelising the load of model versions in one of the PEACEMAKER variants is unfair with respect to EMF Compare and DiffMerge, where a sequential load of the three versions is performed. Our first consideration was that parallelising the load of the other two approaches would not be as beneficial, because three different model files, one for each model version, have to be loaded from disk, which would be done sequentially. On the contrary, PEACEMAKER only loads a single merged model file from disk, and the model versions are obtained by selecting the appropriate lines from in-memory contents (see Fig. 2). So, as the contents are already stored in memory, parallelising the version load is more beneficial for PEACEMAKER. Nonetheless, to confirm this, we performed a parallel load for EMF Compare and DiffMerge for the conflict models of scenario 1, and the results are shown in Fig. 25. We can see that the parallel execution achieved a 16% average reduction

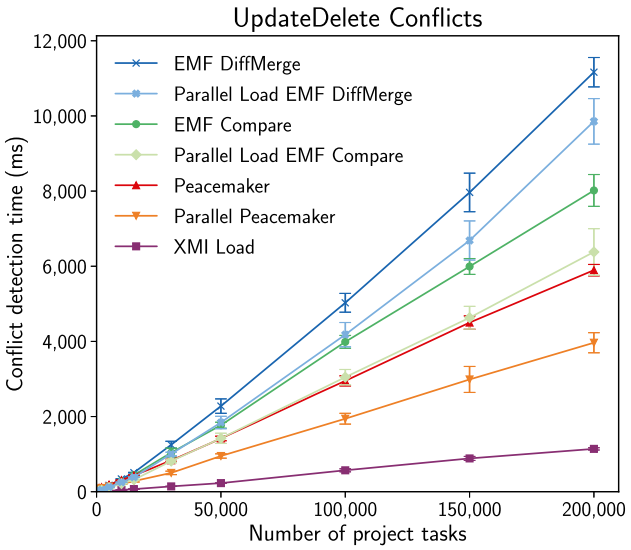

Fig. 25 Detection times of scenario 1 including parallel loads for EMF Compare and DiffMerge

for EMF DiffMerge, while for EMF Compare the reduction was slightly larger at 22%. However, neither of these executions achieved a better result than just the sequential variant of PEACEMAKER, although EMF Compare was able to almost match its times. It is important to also note that our loading times do not take into account the extra work that both EMF Compare and DiffMerge need to perform to gather the model versions from the VCS. Lastly, an inspection of the source code of both EMF Compare and Diffmerge revealed that they do not parallelise the loading of input resources. So, performing sequential version loads for these tools is a better representative of the performance that would be found by end users when actually using the tool. Based on the above, we decided to show the sequential times of all the compared approaches, plus the parallel variant of PEACEMAKER.

5.3 Discussion

In the light of the findings presented in the previous sections, we can now answer the research questions asked in Sect. 1.

5.3.1 RQ1: Can an approach that uses the line-based conflicts marked by a VCS to identify conflicts instead of a full model comparison offer the same results?

Based on the analysis of Sect. 5.1, PEACEMAKER is not able to detect all conflicts that existing tools such as EMF Compare or EMF DiffMerge can find. However, it comes very close to achieving a full detection. The results of Table 2 show that PEACEMAKER is able to properly identify and resolve most conflict cases (45 out of 60), and it is also able to detect conflict smells suggesting potentially missed conflicts by Git (9 out of 60). The remaining 6 cases were automatically merged by Git (identified with the ! symbol), and PEACEMAKER did not detect the underlying conflicts. On the other hand, these automatic merges did not create any immediate issues, such as losing model information or introducing errors that prevent the model from loading.

Table 4 Conflict detection times in ms for some concrete model sizes for Scenario 2 (bottom of Fig. 22)

Tool	Model Size (#elements)		
	5000	15,000	30,000
EMF DiffMerge	165	509	1309
EMF Compare	136	428	1039
PEACEMAKER	97	286	578
Parallel PEACEMAKER	46	135	349

5.3.2 RQ2: Does a conflict detection approach as described in *RQ1* provide better performance and scalability than full model comparison?

The results of Sect. 5.2 show that the two PEACEMAKER variants were consistently faster in all comparison scenarios. For models smaller than $\sim$ 50 K elements, the time it takes all approaches to complete might be too small (less than $\sim$ 2 s) for the improvements of PEACEMAKER to be that noticeable, at least considering an online use of PEACEMAKER (i.e. a user working with the PEACEMAKER editor). On the other hand, being up to about 2 to 3 times faster than the other approaches for the bigger models of the comparison puts PEACEMAKER in an advantageous position in terms of performance, which makes our approach an interesting option to consider in those contexts where large models are causing existing model comparison and merging solutions to suffer from performance issues.

Lastly, we consider that PEACEMAKER could also be useful for smaller models, this is, without these models needing to reach the 50 K elements mark. This mark was selected as a usability measure on tolerable waiting time, which according to [23] the measure sits at about 2 s (i.e. the time some tools need to detect the conflicts at 50 K model elements). Nevertheless, the performance improvements provided by PEACEMAKER are also seen for smaller models, which could be very useful in terms of scalability if several models have to be merged at the same time or if merging of models happens frequently (e.g. in a continuous integration environment).

Table 4 shows the time (in milliseconds) it took EMF DiffMerge, EMF Compare and PEACEMAKER to detect the conflicts of Scenario 2 (Double Update, bottom of Fig. 22) for the PSL models with 5K, 15K and 30K elements. For instance, merging five models with 15 K elements would take around 2.5 s with EMF DiffMerge, 2.1 s with EMF

Compare, 1.4 s with PEACEMAKER, and 0.67 s with parallel PEACEMAKER. It is true that merging several models is a fully parallelisable task (i.e. merging each model in a separate process). However, parallel or not, the total amount of computing work that needs to be carried out is the same, and PEACEMAKER might allow reducing that amount.

5.3.3 RQ3: Is relying on line-based VCSs such as Git to merge XMI models and to detect conflicts safe?

We have demonstrated that a plain line-based merge approach is not safe enough to detect a few types of conflicts that might arise when merging XMI models. We have shown several examples of how false positives (Sects. 3.4 and 5.1.3) and false negatives (Sects. 3.5, 5.1.4 and 5.1.5) can manifest when applying a line-based merge over XMI models. Moreover, this type of merge can introduce errors in the model, e.g. by duplicating lines containing model elements or values.

Therefore, and in line with previous research [13–15], we consider line-based merging alone is unsafe when versioning XMI models in VCSs, both in terms of correctly merging models, and related to detecting all merge conflicts that might occur. However, we have described how PEACEMAKER is able to enhance the identification capabilities of plain line-based merging. Based on our answers to RQ1 and RQ2, it offers a trade-off between identification completeness and performance that might be useful in some contexts, thus making line-based merging viable. We comment on this trade-off in detail in the next section.

5.3.4 When to use PEACEMAKER over model-based approaches

Choosing to merge models using a VCS such as Git along with PEACEMAKER requires tolerating that the conflict types included in the ! group in Table 2 will not be detected. However, not being able to detect this group of conflicts does not cause issues such as losing model information. As detailed in Sect. 5.1.5:

- When a value of a multi-valued attribute is deleted in one of the versions but moved to another position in the other (d_{1attr} case in Table 2), the value is retained in the new position in the merged model.
- When moving an element in one version of the model into a container that is deleted in the other version (*h* category cases in Table 2), the merged model includes both containers.

Tolerating this group of conflicts provides increased conflict detection performance as a trade-off, which could be very reasonable in contexts where large models are updated frequently by teams of engineers with clear responsibilities over different parts of the model.

We consider important to remind the reader that using Git to merge models and only applying a model-based conflict detection tool such as EMF Compare when Git detects a conflict would also imply being oblivious to conflicts of the ! group. The only way a tool such as EMF Compare can also detect these conflicts is by configuring a custom merge strategy in the application used to manage source code repositories (e.g. EGit in the case of the Eclipse IDE), as described in Sect. 4.2.

Other novel features of PEACEMAKER that might be of interest in certain contexts are its support of partial resolution of conflicts (see Sect. 3.6) and its ability to work directly with stand-alone files containing conflict sections, without requiring access to the three model versions in the repository (although PEACEMAKER still needs to know of the required metamodels to which models conform to, and of any external cross-referenced models). This last feature might make easier to deploy PEACEMAKER as part of a continuous integration (CI) process: PEACEMAKER checks take place after the conventional line-based merge, so it is not necessary to substitute that with a custom merge strategy. Instead, PEACEMAKER could be added as an extra validation or test process to execute after the line-based merge to check for issues. This configuration might require less effort than altering the merge strategy of a CI system such as Jenkins to use a model-based tool like EMF Compare or EMF DiffMerge. However, we have not yet performed any validation to support this claim, so we leave this as part of our future work.

Performance issues due to having to merge large models could be avoidable by adopting a model decomposition strategy from the very beginning. Following such a strategy involves splitting models into several fragments according to certain criteria, such as division of labour over them. This kind of strategy is only achievable if the employed modelling tools support model decomposition. Also, splitting models into adequate fragments is not always a trivial task, as collaborative work in cross-cutting concerns might benefit from contrasting/incompatible model split points [24].

The formulated research questions focused on the completeness, performance, and safety of the presented conflict detection approach. The last step to evaluate the exploitability of PEACEMAKER (under the acceptance of the presented compromise between performance and precision) involves carrying out usability experiments, where the PEACEMAKER editor could be tested by final users to determine how well it behaves against alternatives such as EMF Compare or EMF DiffMerge. Usability is not covered by our research questions and/or our evaluation, so it lies outside the scope of this article. Running these experiments will be part of our future work for PEACEMAKER.

5.4 Considerations for merging XMI models with line-based version control systems

Out of the experience gathered during the development of PEACEMAKER, and based on the findings of this evaluation, we include here a set of practices to follow when wanting to keep merging XMI models using line-based VCSs such as Git:

- After merging a model using vanilla Git (e.g. from the command line), it is of utmost importance that the merged model is thoroughly validated with an approach such as PEACEMAKER, as well as with domain-specific model validation processes. We have shown how a clean line-based merge is no guarantee of the model being correctly merged, as it might contain structural issues and it might not conform to its metamodel.
- As mentioned in Sect. 5.1.4, the use of non-unique multi-valued EAttributes can cause duplicated values in line-based merged models under certain conditions and should be avoided whenever possible.
- Auto-incremental identifiers (e.g. *task1*, *task2*, $\cdots$) or ID EAttributes can create issues when adding new elements to a model in two independent development branches. To avoid this, we recommend the use of Universally Unique Identifiers (*UUIDs*) generators, such as the one provided by EMF.[14]

6 Related work

To the best of our knowledge, PEACEMAKER is the first approach that leverages the work done by the standard line-based merge when working with models. The closest approach we have found is the work of Asenov et al. [25], where a line-based approach is presented for two-way comparison (i.e. a *diff* operation [17,18], instead of a merge one) over tree structures. They define a specific syntax to store these trees, which allows using as input the result of a line-based diff to carry out a correct tree comparison. They also present a three-way merge approach over the same tree structure, but that approach works at the level of tree nodes and edges, instead of merging text lines.

Merging operations over text files can be classified according to their use of the inherent structure of the contents of these files. On one side, line-based merging is also denoted as unstructured merging [1,2], because the underlying structure of the data is disregarded. On the other extreme we can find structured merging, also often denoted as syntactical merging [1], where the structure of the data is known and used for the merge operation. This is the kind of operation that is carried out by model merging solutions, such as EMF Compare [11] and EMF DiffMerge [12] and that we described in Sect. 2.4. There are other solutions that apply structured comparison to source code files by comparing their abstract syntax tree (AST), such as Gumtree [26] or ChangeDistiller [27] for the Java programming language. Lastly, there is a third type, known as semi-structured merging [28,29], where a fraction of the file contents are merged based on their structure (e.g. part of the AST of a source code file), and the remaining contents are merged using an unstructured approach. For instance, when merging a Java class, its method signatures can be treated as a tree (i.e. structured merge), while the body of these methods is merged as standard lines (unstructured merge). As for the PEACEMAKER approach, it falls under targeted structured merging, as the XMI files are loaded as models, but the merging operation focuses on those model elements initially marked by the line-based merge, instead of performing a full-model comparison.

Model merging utilities come either bundled within software solutions that target specific modelling technologies, or as standalone third-party applications specifically tailored for the comparison task. For instance, Simulink Models can be merged with the comparison tool provided by MATLAB [30], or by using DiffPlug [31]. Other solutions target general-purpose modelling frameworks, such as IBM Rational Rhapsody [32] for UML; or EMF Compare [11] and EMF DiffMerge [12] for the Eclipse Modelling Framework. This allows reusing the same comparison tools for any domain specific language built atop the concrete framework. Lastly, there are some solutions that are defined generically, but they can be adapted to support specific model technologies. SiDiff [33] and many other academic solutions [15,25,34] are examples of these.

There is also a different family of tools for model versioning that offer specific repositories for the persistence of models. Examples of these tools are Eclipse CDO [35], EMF Store [36], MetaEdit+ [6], MagicDraw [7] or Obeo Designer [8], among others. Developing custom model repositories can provide benefits in different aspects, such as scalability, or a more controlled versioning of models. For instance, change-based persistence techniques [36] could be applied to have a fine-grained control of the changes that are included in models, and to better detect incoming conflicts [37,38]. Another technique that is implemented by several commercial solutions allows avoiding conflicts by locking the whole model or specific parts of the model where a user is currently including modifications [6–8]. This technique requires working against an always-online and centralised model server, so that locking information can be distributed among all concurrent users. While specific model repositories offer interesting features, such repositories are typically proprietary, re-implement similar functionality (user management,

[14] https://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/util/EcoreUtil.html#generateUUID().

model fragment locking/unlocking, check-in/out), and lack in features such as branching and tagging. In addition, these repositories need to be administered separately from the VCS used to store any source code, and there is limited tool support for them outside the modelling environments for which they were initially developed for (e.g. integration with other IDEs, continuous integration systems, and other third-party model measurement and analysis tools). Finally, they arguably lack in robustness compared to file-based VCSs such as Subversion and Git.

7 Conclusions and future work

We have introduced a new approach for the detection and resolution of conflicts that works against merged XMI files with conflict sections produced by mainstream text-based VCSs like Git. This approach, which has been implemented in the EMF-based PEACEMAKER tool, can detect almost the same types of conflicts than standard model-based approaches are able to, while taking up to 60% less time to do so. The undetected conflicts do not cause any loss of model information, which makes PEACEMAKER an option to consider in those contexts where missing these conflicts is tolerable and extra performance is required.

For future work, we plan to test PEACEMAKER with end users to polish any issues related to the usability of the editor. In addition, we wish to test if a similar line-based approach could be valid to also speed up 2-way model comparison, i.e. by parsing the added and deleted lines of a *diff* command over the two compared versions. We would also like to study whether PEACEMAKER could be integrated easily with external version control tools and continuous integration environments, either by adding a custom merge strategy for models or by including an extra validation step where PEACEMAKER is used to analyse the resulting models after a line-based merge.

Acknowledgements We would like to thank the developers of EMF Compare for making their tests suites available, which were very useful during the development and evaluation of PEACEMAKER.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

References

1. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. **28**(5), 449–462 (2002). https://doi.org/10.1109/TSE.2002.1000449
2. Khanna, S., Kunal, K., Pierce, B.C.: A formal investigation of diff3. In: FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, pp. 485–496. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77050-3_40
3. Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: An introduction to model versioning. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 7320 LNCS, pp. 336–398 (2012). https://doi.org/10.1007/978-3-642-30982-3_10
4. Paige, R.F., Matragkas, N., Rose, L.M.: Evolving models in model-driven engineering: state-of-the-art and future challenges. J. Syst. Softw. **111**, 272–280 (2016). https://doi.org/10.1016/j.jss.2015.08.047
5. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional (2009)
6. Kelly, S.: Collaborative modelling with version control. In: Software Technologies: Applications and Foundations, pp. 20–29. Springer (2018). https://doi.org/10.1007/978-3-319-74730-9_3
7. No Magic: MagicDraw. https://www.nomagic.com/products/magicdraw
8. Obeo: Obeo Designer. https://www.obeodesigner.com/
9. Object Management Group: XML Metadata Interchange Specification. https://www.omg.org/spec/XMI/
10. Eclipse Foundation: Sirius. https://www.eclipse.org/sirius/
11. Eclipse Foundation: EMF Compare. https://www.eclipse.org/emf/compare/
12. Eclipse Foundation: EMF DiffMerge. https://www.eclipse.org/diffmerge/
13. Barrett, S., Chalin, P., Butler, G.: Model merging falls short of software engineering needs. In: Proceedings of the 2nd Workshop on Model-Driven Software Evolution (2008)
14. Altmanninger, K., Brosch, P., Langer, P., Seidl, M., Wiel, K., Wimmer, M.: Why model versioning research is needed!? An experience report. In: MoDSE-MCCM Workshop in MoDELS, pp. 1–12 (2009)
15. Schwägerl, F., Uhrig, S., Westfechtel, B.: A graph-based algorithm for three-way merging of ordered collections in EMF models. Sci. Comput. Program. **113**, 51–81 (2015). https://doi.org/10.1016/j.scico.2015.02.008. (Model Driven Development (Selected & extended papers from MODELSWARD 2014))
16. Object Management Group: Meta Object Facility (MOF) Core Specification. https://www.omg.org/spec/MOF/ (2016)
17. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Commun. ACM **20**(5), 350–353 (1977). https://doi.org/10.1145/359581.359603
18. Miller, W., Myers, E.W.: A file comparison program. Softw. Pract. Exp. **15**(11), 1025–1040 (1985). https://doi.org/10.1002/spe.4380151102
19. Somogyi, F.A., Asztalos, M.: Systematic review of matching techniques used in model-driven methodologies. Softw. Syst. Model. **19**(3), 693–720 (2020). https://doi.org/10.1007/s10270-019-00760-x
20. SAX Project: Simple API for XML. http://www.saxproject.org/

21. López, J.A.H., Cuadrado, J.S.: Towards the characterization of realistic model generators using graph neural networks. In: 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 58–69 (2021). https://doi.org/10.1109/MODELS50736.2021.00015
22. Cooper, J., de la Vega, A., Paige, R.F., Kolovos, D.S., Bennett, M., Brown, C., Piña, B.S., Rodriguez, H.H.: Model-based development of engine control systems: Experiences and lessons learnt. In: 24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021, Fukuoka, Japan, October 10–15, 2021, pp. 308–319. IEEE (2021). https://doi.org/10.1109/MODELS50736.2021.00038
23. Nah, F.F.H.: A study on tolerable waiting time: how long are web users willing to wait? Behav. Inf. Technol. **23**(3), 153–163 (2004). https://doi.org/10.1080/01449290410001669914
24. Bendix, L., Emanuelsson, P.: Diff and merge support for model based development. In: Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models—CVSM'08, p. 31. ACM Press, Leipzig, Germany (2008). https://doi.org/10.1145/1370152.1370161. http://portal.acm.org/citation.cfm?doid=1370152.1370161
25. Asenov, D., Guenat, B., Müller, P., Otth, M.: Precise version control of trees with line-based version control systems. In: Fundamental Approaches to Software Engineering (FASE), pp. 152–169. Springer, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_9
26. Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE'14, pp. 313–324. Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2642937.2642982
27. Fluri, B., Wursch, M., PInzger, M., Gall, H.: Change distilling: tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. **33**(11), 725–743 (2007). https://doi.org/10.1109/TSE.2007.70731
28. Apel, S., Liebig, J., Brandl, B., Lengauer, C., Kästner, C.: Semistructured merge: Rethinking merge in revision control systems. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE'11, pp. 190–200. Association for Computing Machinery, New York (2011). https://doi.org/10.1145/2025113.2025141
29. Cavalcanti, G., Borba, P., Seibt, G., Apel, S.: The impact of structure on software merging: semistructured versus structured merge. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1002–1013. IEEE (2019). https://doi.org/10.1109/ASE.2019.00097
30. MathWorks: Merge Simulink Models. https://uk.mathworks.com/help/simulink/ug/merge-simulink-models-from-the-comparison-report.html
31. DiffPlug: Simulink Diff. https://www.diffplug.com/features/simulink
32. IBM: Rational Rhapsody DiffMerge. https://www.ibm.com/support/knowledgecenter/SSB2MU_8.2.0/com.ibm.rhp.diffmerge.doc/topics/rhp_c_col_parallel_dev_with_diffmerge.html
33. Schmidt, M., Gloetzner, T.: Constructing difference tools for models using the sidiff framework. In: W. Schäfer, M.B. Dwyer, V. Gruhn (eds.) 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10–18, 2008, Companion Volume, pp. 947–948. ACM (2008). https://doi.org/10.1145/1370175.1370201
34. Alanen, M., Porres, I.: Difference and Union of Models. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 2863, pp. 2–17 (2003). https://doi.org/10.1007/978-3-540-45221-8_2
35. Eclipse Foundation: Eclipse CDO. https://www.eclipse.org/cdo/
36. Koegel, M., Helming, J.: Emfstore: A model repository for EMF models. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 2, ICSE'10, pp. 307–308. Association for Computing Machinery, New York (2010). https://doi.org/10.1145/1810295.1810364
37. Koegel, M., Helming, J., Seyboth, S.: Operation-based conflict detection and resolution. In: 2009 ICSE Workshop on Comparison and Versioning of Software Models, pp. 43–48. IEEE (2009). https://doi.org/10.1109/CVSM.2009.5071721
38. Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: A fundamental approach to model versioning based on graph modifications: from theory to implementation. Softw. Syst. Model. **13**(1), 239–272 (2014). https://doi.org/10.1007/s10270-012-0248-x

Alfonso de la Vega is an Assistant Professor at the University of Cantabria. Previously, he was a Research Associate working at the University of York. He collaborates as an Eclipse Foundation Committer for the Epsilon project. His more recent research focuses on novel model visualisation and comparison approaches. He has also worked in how to apply modelling and domain-specific languages to reduce the complexity of carrying out data engineering and data mining tasks.

Dimitris Kolovos is a Professor of Software Engineering in the Department of Computer Science at the University of York, where he researches and teaches automated and model-driven software engineering. He is also an Eclipse Foundation committer, leading the development of the open-source Epsilon model-driven software engineering platform, and an editor of the Software and Systems Modelling journal. He has co-authored more than 150 peer-reviewed papers and his research has been supported by the European Commission, UK's Engineering and Physical Sciences Research Council (EPSRC), InnovateUK and by companies such as Rolls-Royce and IBM.