Galaxy : Developpement collaboratif de systèmes complexes selon une approche guidée par les modèles

Deliverable D2.2: Mechanism for Collaborative Unit Synchronization

	NAME	PARTNER	DATE
WRITTEN BY	J. Robin	LIP6	09/12/2010
	X. Blanc	LIP6	09/12/2010
REVIEWED BY			

RECORD OF REVISIONS

ISSUE	DATE	EFFECT ON		REASONS FOR REVISION
		PAGE	PARA	
01	15/10/2010			Création du document
02	28/10/2010			Prise en compte de la réunion Galaxy Toulouse
03	03/12/2010			Prise en compte de la réunion Galaxy Bordeaux
04	09/12/2010			Finalisation première version SVN

TABLE OF CONTENTS

TABLE OF APPLICABLE DOCUMENTS

N°	TITLE	REFERENCE	ISSUE	DATE	SOURCE	
					SIGLUM	NAME
A1						
A2						
A3						
A4						

TABLE OF REFERENCED DOCUMENTS

N°	TITLE	REFERENCE	ISSUE
R1	Galaxy glossary		
R2	Collaborative Unit Definition	D2.1	

1. INTRODUCTION

This document presents D2.2., *i.e.* the second deliverable of the second work package of the project. The goal of this deliverable is to detail three key aspects in the *Collaborative Unit (CU)* specification presented in D2.1, the preceding deliverable of the same work package.

As defined in the D0.1.1 deliverable presenting the Galaxy project, these three aspects are the following:

1. *Collaborative unit diff*: a specification of the collaborative unit operation that compares different versions of a given artifact set stored in the same or two different collaborative units; this operation must return all the differences between the artifact set pair passed to it as input; it constitutes the first main aspect of the collaborative unit synchronization mechanism;
2. *Collaborative unit merge*: the specification of one strategy to merge two versions of a given artifact set stored in the same or two different collaborative units; this operation must return an artifact set that contains all the compatible model elements that these artifact sets persistently store; it constitutes the second main aspect of the collaborative unit synchronization mechanism; note that if some model element in the two artifact set versions to merge are incompatible, the merge operation fails; specifying it thus also entails defining one strategy to detect incompatibility between elements;
3. *Means to balance the collaboration strategy*: there are two main collaborative development schemes (a) the lock-modify-unlock sequence, called pessimist locking and (b) the copy-modify-merge sequence, called optimist locking; depending on a variety of project specific factors, such participant organization policies, modification granularity and frequency, coupling level between artifacts, these two strategies may have significantly different relative performance; insuring scalable interaction thus requires to support both approach; since this is the case of the collaborative unit concept as defined in D2.1., this aspect was in effect already address in D2.1 with no need for further elaboration in the present D2.2.

The document is organized as followed. In section 2, we quickly review the concepts defined in the preceding deliverable D2.1 that the present D2.2 elaborates. In section 3 and 4, we then present in turn the CU *diff* and the CU *merge*. For each concept, we first recall their usage and define the principles on which they are based. We then provide the new data structures needed to add into the Galaxy framework to specify, intuitively in natural language, the step-by-step algorithm for the corresponding operation (*diff* or *merge*). Finally, in section 5, we provide a small illustrative example of each operation call and result.

In Appendix 1, we give one precise procedural specification of the *diff* operation body as expressions in IOCL (Imperative Object Constraint Language).

2. REMINDER ON THE GALAXY FRAMEWORK

2.1 LAYERED MVC ARCHITECTURE

D2.1 structures the Galaxy framework[1] in three layers:

[1] In this document and in the previous deliverable D2.1, we use the (albeit overloaded) word "framework" to mean an object-oriented *conceptual* framework made of abstract classes, interfaces and concrete classes that define general concepts. It does not constitute an *architectural* framework which is to be defined in WP4, based on some of these concepts.

The Galaxy framework *API*, shown in

1. Figure 1; it defines a set of conceptual interfaces through which the CASE tool of each development team can interoperate;
2. The Galaxy *artifacts* shown in Figure 2 ; it defines the conceptual data structures of models, model elements and model views;
3. The Galaxy *collaborative unit* shown in Figure 3; it defines an intermediary layer that mediates between the high-level Galaxy framework API layer and the low-level Galaxy artifact layer.

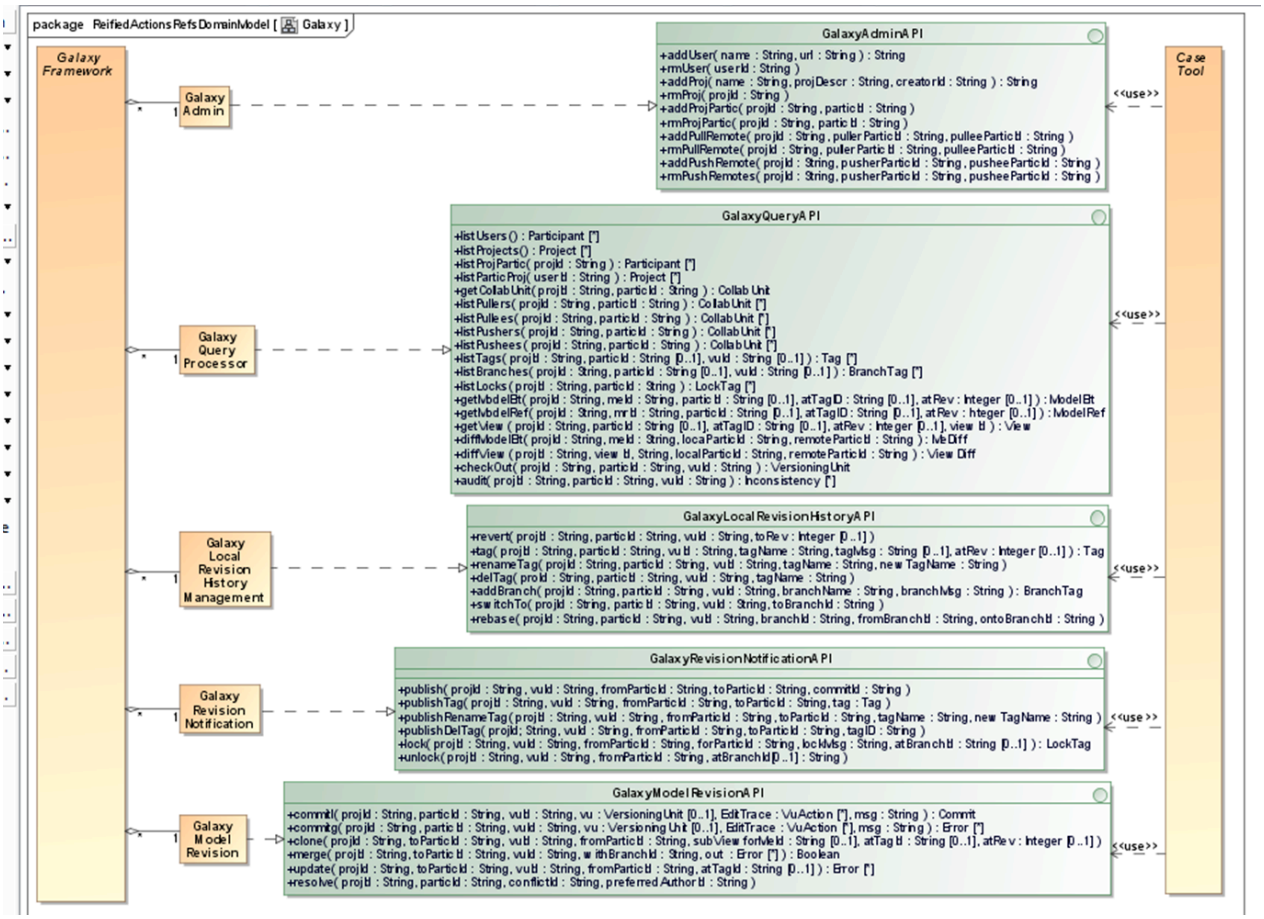

Figure 1 : The Galaxy framework API (from D2.1)

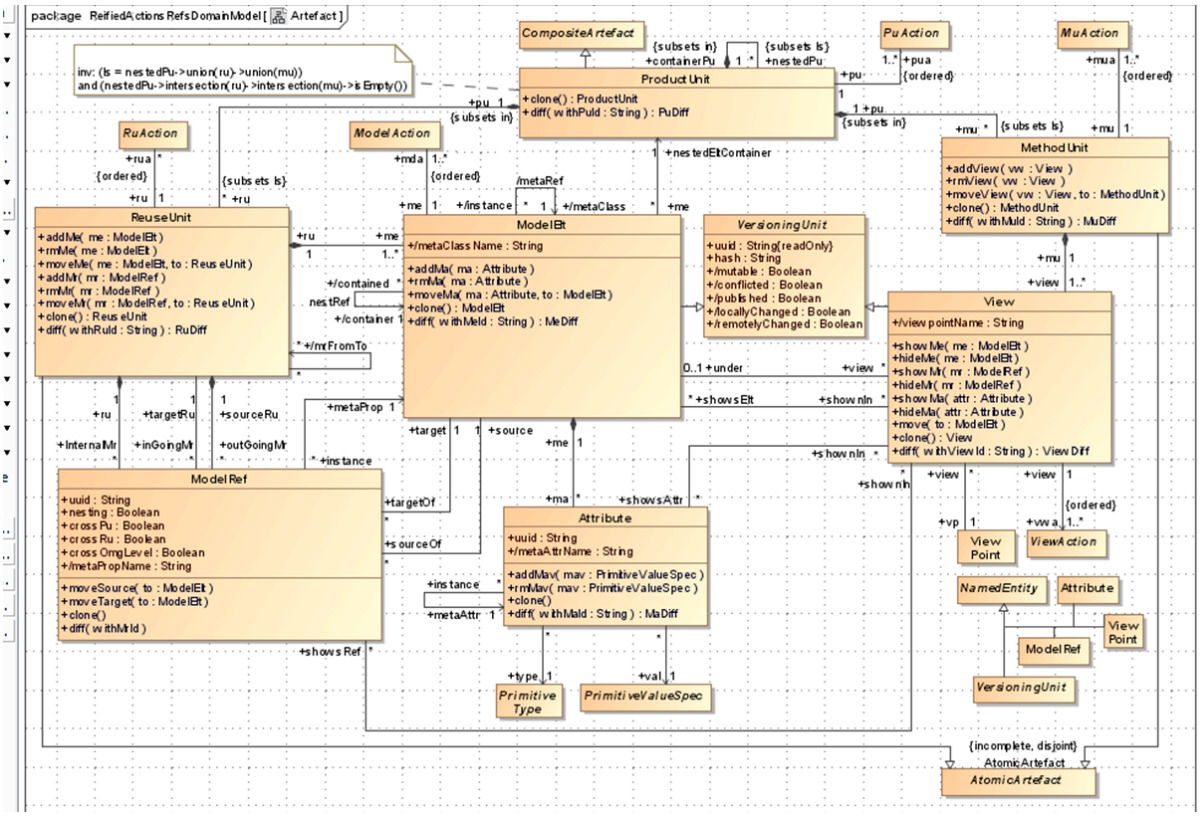

Figure 2 : The Galaxy artifacts (Product, Reuse and Method Units, from D2.1)

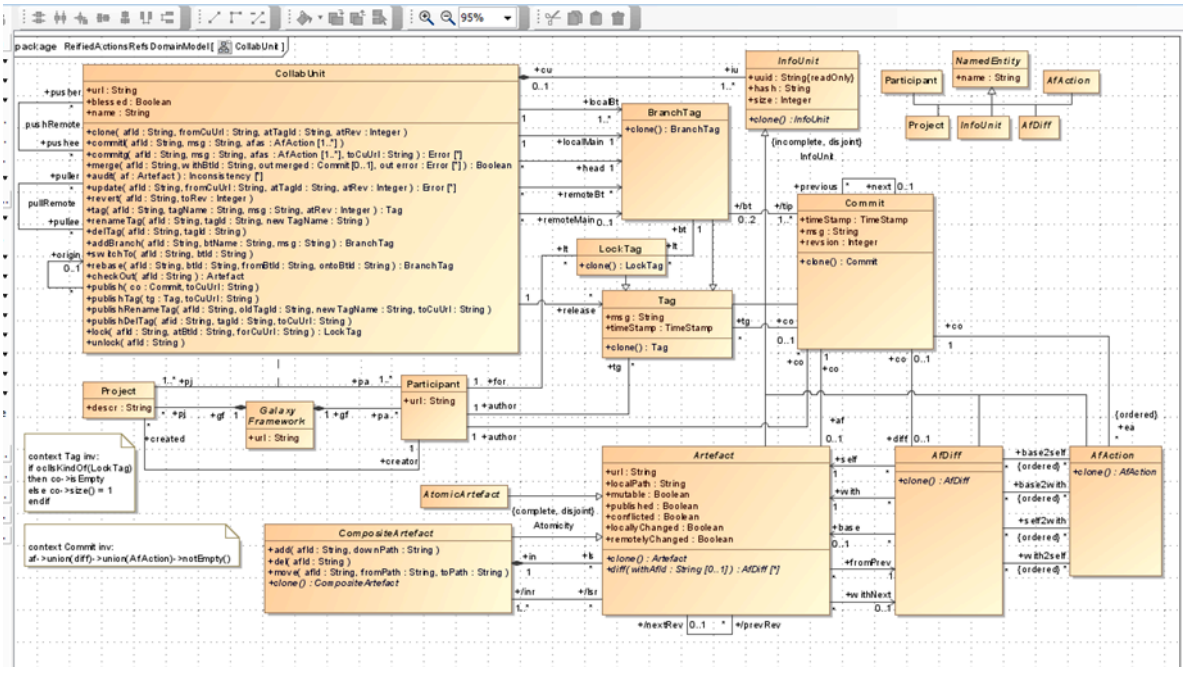

Figure 3 : The Galaxy collaborative unit (from D2.1)

2.2 MODEL FRAGMENTATION

As shown in Figure 2, D2.1 defined three classes of "artifacts": Product Unit (PU), Reuse Unit (RU) and Method Unit (MU). PU form an artifact containment tree which leaves are RU and MU. The model is partitioned into the RU. The set of all model views is partitioned into the MU.

As shown in Figure 3, each project participant possesses its own Collaborative Unit (CU) which groups the artifacts containing the model elements and views relevant to him (her).

As shown in

Figure 1, all four classes of units (CU, RU, MU and PU) are hidden for the participant (*i.e.*, Galaxy framework user). Participants only know that they are working on model elements, shown into views (*e.g.* UML diagrams).

Figure 4 shows a simple example with a participant collaborating with others on a model that contains five elements shown in two views. The participant has a CU that contains one PU that itself contains one RU that stores all the model elements and two MU (one for each view).

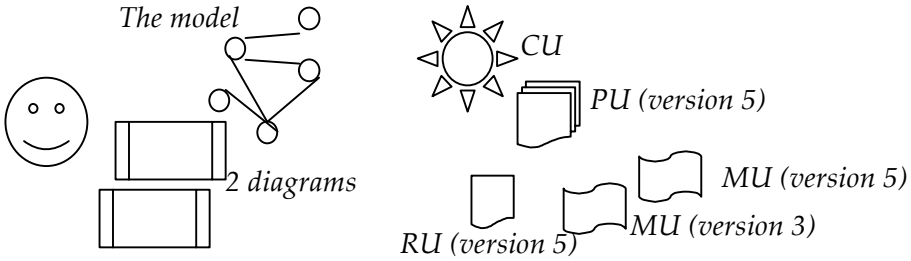

Figure 4 : The CU of a project participant gathers the PU, RU and MU containing the model elements and views (diagrams) onto which (s)he collaborates.

As shown in Figure 3, for each project, there is one Galaxy CU per participant and one Galaxy CU per global blessed repository.

When a project follows the *copy*, *modify*, *merge* collaboration paradigm, synchronizing the local and the remote collaborative units requires the availability of a *merge* operation between model elements, model views and the artifact storing them (PU, RU and MU).

2.3 FRAMEWORK CONFIGURATION AND REVISION STRATEGY

As shown in Figure 5, the Galaxy framework is configurable by what we called a *revision strategy*. Such strategy defines:

1. How the model is fragmented (model fragmentation strategy) by defining (1) how model elements are partitioned into product and reuse units, (2) how model views are partitioned

into product and method units and (3) how model and view revision actions are translated at the CU layer into artifact revision actions (*i.e.*, PU, RU and MU act);

2. How the pairs of different model elements and model views are merged into one;
3. How merged elements and views are audited for inconsistencies.

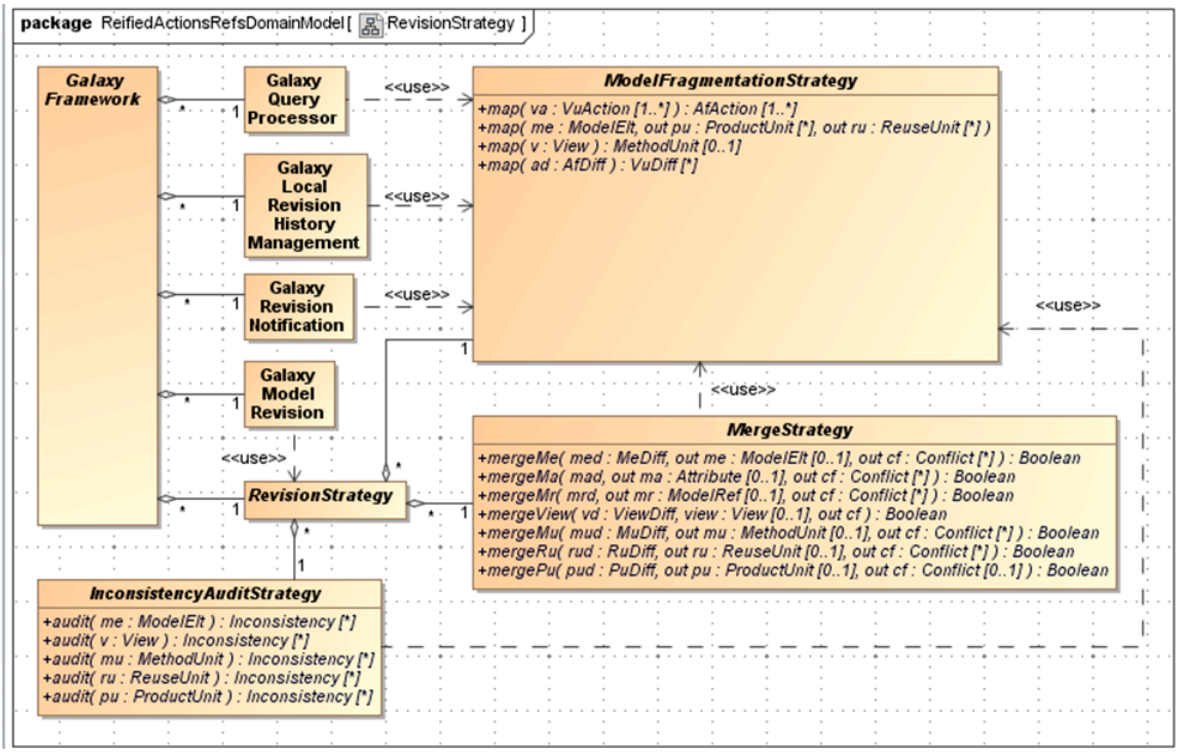

Figure 5 : Configurating the Galaxy framework with a revision strategy (from D2.1)

To use the Galaxy framework as revision control system for a collaborative MDE project a team must first configure it by defining the *ModelFragmentationStrategy*, *MergeStrategy* and *InconsistencyStrategy*. In deliverable D2.1, we gave several examples of concrete model fragmentation strategies. In the present deliverable D2.2, we give one example of concrete merge strategy. We also explain how an inconsistency strategy is used in concert with the merge strategy in the basic revision control operations *commitg* and *update* of the framework. These operations were specified in D2.1 deliverable, we summarize their role, as well, as their relation with the operations *diff* and *merge*, focus of the D2.2.

2.4 COMMIT AND UPDATE

After a participant locally changes model elements or views by using a CASE tool connected to a Galaxy framework, (s)he needs to commit these changes to a CU to make them available to participants with whom (s)he collaborates. In this short reminder, we will suppose, for the sake of simplicity, that the framework is configured in centralized mode. In such mode, a participant action results in a call to the *commitg* operation of the Galaxy framework API to commit its local changes to a remote blessed CU. As argument, the *commitg* of the API takes as main input either a model element or a view element.

The Galaxy framework assumes that when selecting one model element to be committed, the participant in fact wishes to commit not only this element (called the root element), but together with it all the elements located below it in the element containment tree; *commitg* calls the *map* operations of the model fragmentation strategy on both the root element selected by the participant and all its descendants in the containment tree.

Depending on the model fragmentation strategy implemented by these map operations, these descendants may be located in different PUs and RUs. Therefore, after calling those map functions, the *commitg* operation of Galaxy model revision class, calls the *commitg* of the CU for all artifacts (PU or RU) that contains the root and its descendant elements. This *commitg* call increments the *revision* attribute (an integer) of these artifacts.

After this increment, the *remotelyChanged* attribute of the root artifact (PU) containing all these elements (and those referenced by them) is read. If it is false, the remote CU contains the same version than the one contained in the local CU before the local changes. A new commit object is then simply created in the remote CU history. This object points to a copy of the artifacts resulting from the committed local changes.

If in contrast, the value of *remotelyChanged* is true, it means than remote changes occurred concurrently with the local ones. In this case, the *commitg* operation fails and the participant needs to call the *update* operation before attempting to commit again the local changes. It is this *update* operation that calls the *merge* operations defined in the merge strategy to try automatically merge those two concurrent set changes: the local one and the remote one. It is thus the need for reconcile concurrent changes that motivates the *merge* operations.

Automatic merge only succeeds when the concurrent local and remote changes concerned unrelated model elements, references and attributes. When the changes interfere in a conflicting way, the merge fails. In such case, it returns a conflict (class *Conflict*) instead of returning a merged element (or branch, view or artifact). As we will explain in section 4, a conflict occurs when the local and remote changes cannot be reconciled into a single well-formed labeled directed graph.

Even if such purely syntactic reconciliation is possible, the merge operation does not return before calling the *audit* operation of the inconsistency audit strategy on the reconciled well-formed labeled direct graph, To goal of this audit call is to check whether this graph does not violate consistency constraints (*e.g,* those specified in a meta-model). If it does violate one such constraint, the merge then fails and returns the inconsistencies that caused the failure instead of the reconciled graph.

When a *merge* operation fails, the *update* operation that called it also fails. This is when the participant who called the *update* operation needs to call the *diff* operation. This operation compares the elements resulting from the local changes, from those resulting from the concurrent remote changes. It explicitly displays all the differences between the two. This allows the participant to understand the causes of the error returned by the failed *merge* call, correct them manually using a CASE tool, and call *commitg* again on the correction.

3. DIFF

3.1 PRINCIPLES

At the Galaxy framework layer, a participant calls the *diffModelElt* operation (resp. *diffView*) operation to compare two versions of one model element (resp. one model view).

The principles of *diffModelElt* are the following. First, this operation does not merely compare the versions of the single model element *rootMe* passed as input argument. Instead, it recursively compares:

- all the descendants *down(rootMe)* of rootMe in the model element containment tree;
- all the elements stored in the same RU *sameRu(down(rootMe))* than these descendants;
- all the elements stored in the RU set *ref(sameRu(down(rootMe)))* where some reference from or to *sameRu(down(rootMe))* was locally changed;

Recursing on the model element containment tree is motivated by a user-interface concern. When a participant wishes to compare two versions of a model element, (s)he generally implicitly means to compare not only the attributes of these elements but also those of all its nested elements.

Recursing on other elements stored in the same or referenced artifacts than the one to compare is also motivated by a user-interface concern: the model fragmentation strategy that partitions elements in artifacts must remain transparent to the participant.

This transparency may lead to the following situation for a participant *lp*:

- participant *lp* calls *commitg* on a model element *rootMe*;
- it fails, because while *lp* was changing *rootMe* locally, another participant *rp* concurrently changed another element *pointingMe* and committed it to the blessed CU before *lp*,

unfortunately both *rootMe* and *pointingMe* belongs to the same RU. As a consequence, while rp has committed *pointingMe*, the version of RU has been increased.

- comparing the local and remote versions of *down(rootMe)*, thus requires comparing all the artifacts that were changed by *rp*'s commit.

The consequence of the first principle is thus that the *diffModelElt* operation compares two versions of a model element set. These two versions, a *local* one and a *remote* one, concurrently evolved from a *base* version, their common ancestor in the revision history.

The second principle of the *diffModelElt* operation (resp. diffView) is that it proceeds in three main steps. The first compares the local version with the base version. The second compares the remote version with the base version. These two steps are asymmetric comparisons of model elements. They return the elements that have been created (or deleted) and the attributes and the references that have been assigned from the base version. The third step makes a symmetric comparison on the results of the first two steps. It returns a set of mismatches. In the next subsection, we give and explain a precise data model for such mismatches.

3.2 DIFF DATA STRUCTURES

The data structure representing both the input and output of the operations *diffModelElt* and *diffView* are shown in Figure 6. The former returns an object of class *MeDiff* (Model Element Diff) while the latter returns an object of the class *ViewDiff.*

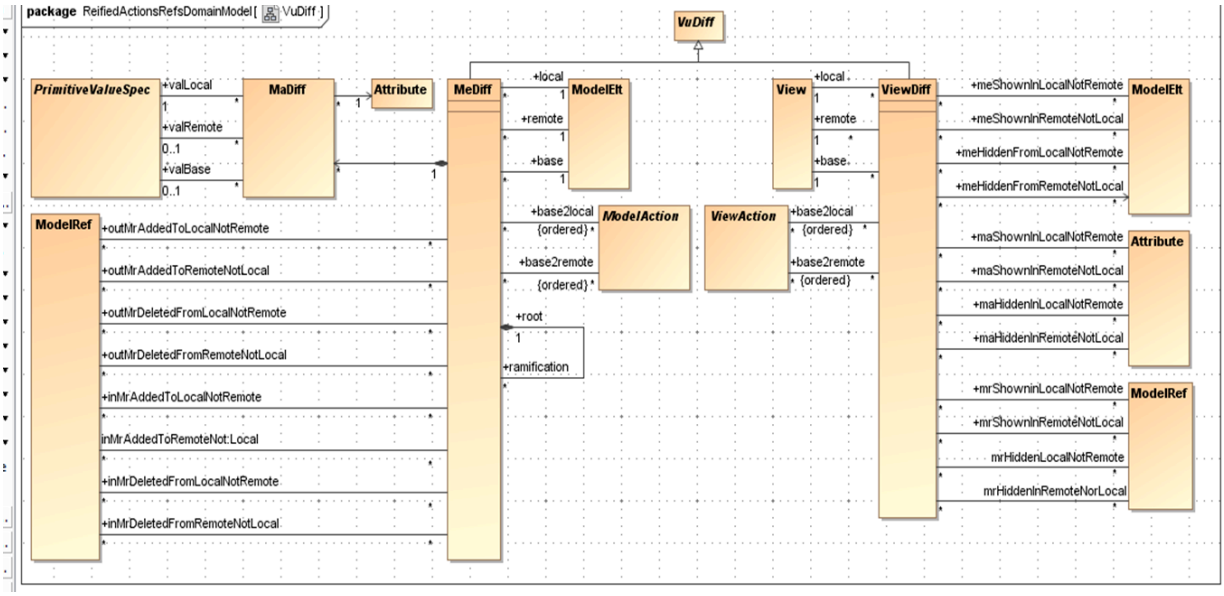

Figure 6 : Diff data structures

A *MeDiff* defines the differences that exist between three versions (local, remote, base) of a same model element. *MeDiff* lists the primitive values that differ in the *local*, *remote* and *base* versions (class *MaDiff*, i.e., Model Attribute Diff). *MeDiff* also lists all the references (objects of class *ModelRef*) ingoing to or outgoing from the element, that were added or deleted from the *base* to the *local* version but not from the *base* to the *remote* version, or vice-versa from the *base* to *remote* version but not from the *base* to the *local* version.

An *MeDiff* object recursively contains other *MeDiff* objects (auto-composition from the *root* role to the *ramification* role).

3.3 DIFF ALGORITHMS

3.3.1 Model Element Diff Algorithm

As input, *diffModelElt* take two versions, *local* and *remote* of the same model element. From these versions it has access to their common ancestor *base* version in the revision history. As input, *diffModelElt* may also have access to:

- the local sequence of model actions (role *base2local*) that changed the base version into the local version;
- the remote sequence of model actions (role *base2remote*) that changed the base version into the remote version.

This is the case if the *commitg* operation stores these actions in the RU of the committed element (role *mda* navigating from *ModelElt* class to the *ModelAction* class shown in Figure 2).

If it is not the case, such sequences can be computed from the local, remote and base elements following algorithms described in [9] (diff by ids algorithm).

3.3.2 View diff algorithm

As input, *diffView* takes two versions, *local* and *remote* of the same model view. From these versions it has access to their common ancestor *base* version in the revision history. As input, *diffViewt* may also have access to:

- the local sequence of view actions (role *base2local*) that changed the base version into the local version;
- the remote sequence of view actions (role *base2remote*) that changed the base version into the remote version.

This is the case if the *commitg* operation stores these actions in the MU of the committed view (role *vwa* navigating from the *View* class the *ViewAction* class shown in Figure 2). There are only six

subclasses of *ViewAction: showMe*, *hideMe*, *showMr*, *hideMr*, *showMa* and *hideMa*, to show in the view or hide from the view a given a model element, a model element attribute or a model reference. Since these actions are all trivial switches without any recursion or iteration, computing them from the local and base views is straightforward. The ouput *ViewDiff* of the *diffView* operation simply lists the model elements (resp. model attribute, model references) that are shown (resp. hidden) in the local version of the view but not in the remote, and vice-versa.

4. MERGE

4.1 PRINCIPLES

A merge is used when changes are performed concurrently. The main goal is to return one model that integrates almost all the changes.
A merge has to analyze the differences between the changes in order to identify if they are in conflict or not. A conflict is a pair of incompatible changes. For instance, if one change consists in removing a model element while another consists in assigning a property value to the element, then those two changes are in conflict. Only one of them can be integrated in the resulting model. Regarding all non-conflict changes, all of them can be integrated. However, the merge algorithm can choose not to consider some of them.

4.2 ERROR, CONFLICT AND INCONSISTENCY

In our context, changes are either expressed by ModelDiff or ViewDiff. A ModelDiff (resp. ViewDiff) targets only one element and represents all the differences between the three versions (base, remote or local) of the element.

Looking at the differences of a ModelDiff, the following rules define whether the changes that apply to the element of ModelDiff are in conflict:

- If a same attribute is modified with two different values in the remote and the local version, then there is a conflict.
- If the element is deleted in one version (local or remote) and modified in the other version (remote or local), then there is a conflict.

Looking at the differences of a ViewDiff, the following rules define whether the changes that target the element are in conflict:

- If an attribute of an element is showed in one version (local or remote) and the element is hidden in another version (remote or local), then there is a conflict.
- If a reference of an element is showed in one version (local or remote) and the element is hidden in another version (remote or local), then there is a conflict.

4.3 ALGORITHM

In this section, we propose one possible Merge algorithm. It should be noted that any other algorithm can be substituted to this algorithm. This algorithm considers that all non-conflict changes are integrated in the returned model.

4.3.1 Model Conflict

The following rules apply to merge the two possible kinds of conflict:

- If the conflict is related to the attribute values, then the value of the local version is used in the returned model.
- If the conflict is related to the deletion of a model element, then the element is not deleted in the returned model and the other change is intergrated.

4.3.2 View Conflict

The following rule applies to merge the different kinds of conflict:

- The show operation is integrated whereas the hid operation is not.

5. ILLUSTRATIVE EXAMPLE

This section presents a simple example that highlights how collaborative units are synchronized. In this example, two developers (Developer A and Developer B) collaborate to build a same model. They use a global blessed repository to synchronize their work. They both share a same **model fragmentation revision strategy** (see D2.1) that consists in storing each model element in its own Reuse Unit (RU). Moreover, for each Reuse Unit, there is only one Product Unit that contains it. On top of that, a single root Product Unit groups all Product Units. The Figure 7 presents the icons we use to present this example.

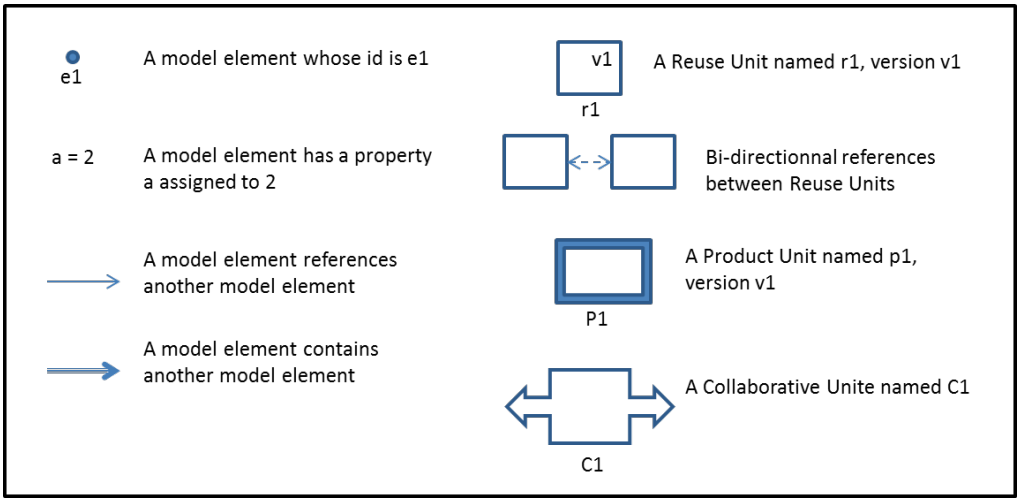

Figure 7 : icons used to presents model elements and units

The Figure 8 presents the initial model. This model is composed of two model elements (e1 and e2). There is a link between e1 and e2, which is a containment link (i.e. the Association that types this link is an composition). According to the strategy, each model element has its own RU (ru1 and ru2). Moreover, each RU has its own PU (pu1 and pu2). On top of that, all PUs are contained in the root PU (pu0 is the root PU). The blessed Collaborative Unit contains all the units.

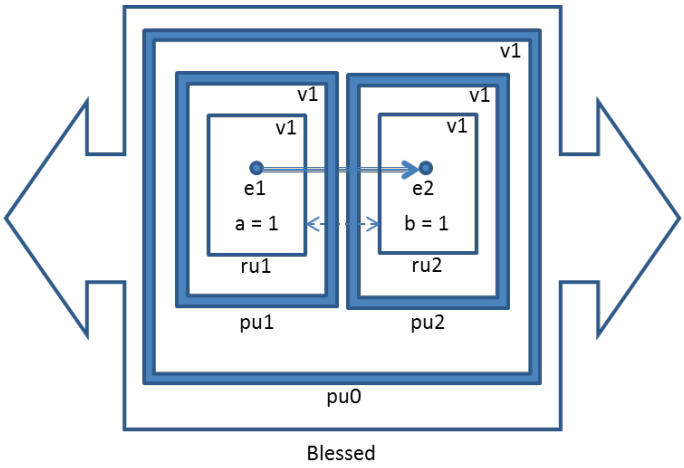

Figure 8 : Initial model (in the blessed CU)

In our example, the two developers (developer A and developer B) update their Collaborative Units in order to start to collaborate.

A modifies the model. He removes the link between e1 and e2. Then he deletes e2. Then he creates a new model element (e3) and assigns a link from e1 to e3. Figure 9 presents the Collaborative Unit of developer A after performing those modifications. According to the strategy, a reuse unit has been created to store e3. Moreover, since e1 and e2 have changed, the versions of ru1, ru2 and their containing PUs have been increased.

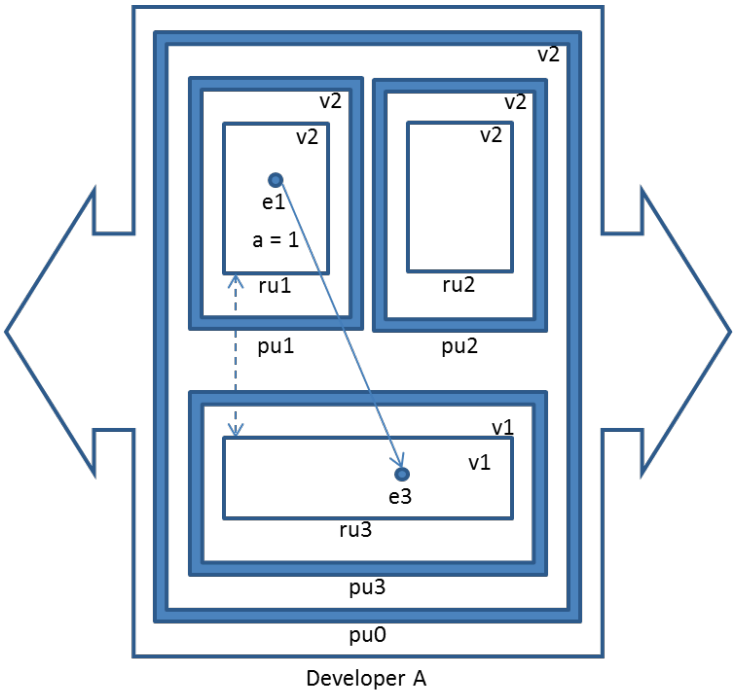

Figure 9 : CU of developer A, after modification

Now, imagine that developer A commits his work to the blessed collaborative unit. He selects e1 as the root element to commit. The algorithm finds that ru1 has to be committed but also ru2 (as a reference has been removed between ru1 and ru2) and ru3 (as e1 references e3). As the versions of those RUs match the versions of the corresponding RU stored in the blessed CU, the commit is accepted. Hence, the blessed CU and the CU of developer A are synchronized.

Concurrently, the developer B modifies the model. He changes the assigned value of e1::a (a=4) and the value of e2::b (b=3). Figure 10 resents the collaborative unit of developer B. As e1 and e2 have been changed, the versions of their RU have been increased.

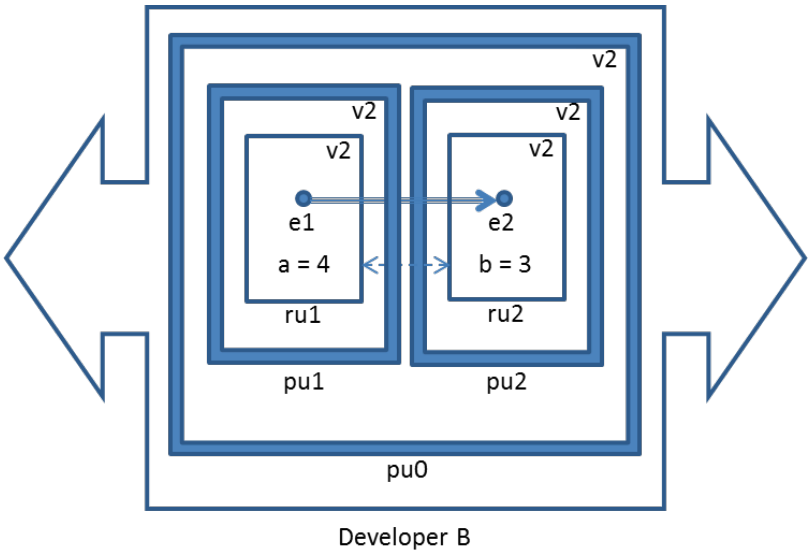

Figure 10: The CU of developer B, after the changes

Now, imagine that developer B wants to commit his work to the blessed collaborative unit. He selects e1 as the root element to commit. The algorithm finds that ru1 and ru2 has to be committed (both e1 and e2 have been changed). As the versions of those RUs mismatch the version of the corresponding RU stored in the blessed CU, the commit is not accepted.

Developer B then decides to update his work and to call the diff algorithm. The diff algorithm yields that conflictual changes have been made. Regarding e1, the property "a" is either assigned to 3 (by developer A) or to 4 (by developer B). Regarding e2, either it is removed (by developer B) or its property is changed (b=3). Other changes are not conflicts (the link between e1 and e2 has been removed; e3 has been created and linked with e1).

B then manually changes the model. He keeps the change of the developer A for e1 (a=3) but keeps his change for e2 (e2 is not deleted and b=3). Figure 11 presents the final Collaborative Unit that is committed to the blessed Collaborative Unit. Developer B and the blessed Collaborative Unit are now synchronized. Developer A will be synchronized when he will update his model.

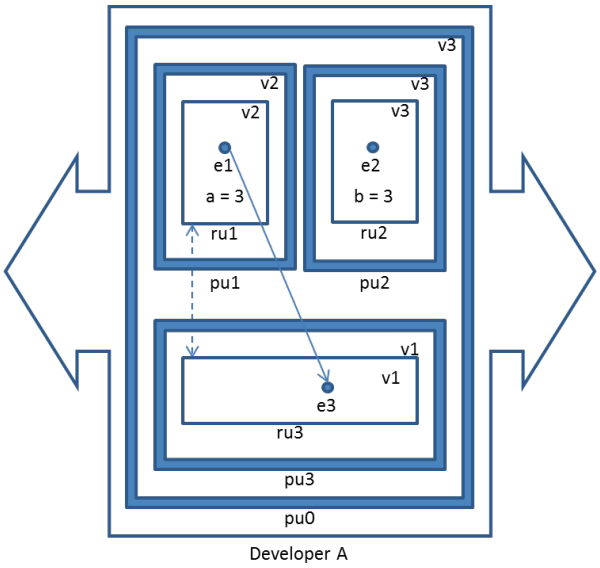

Figure 11: Changes commited by the developer A, after conflicts resolution

6. DISCUSSION ON SCALABITY

Regarding scalability, the less PU, MU and CU are modified in reaction to committing a model element, the less conflict may occur. Section 3 clearly explains that a set of PU, MU and RU are considered while committing one model element. Those PU, MU and RU contain model elements others than the one that has been changed. At least, they contain the containment tree of the root model element.

The first principle that has to be respected is the fact that the number of considered PU, MU and RU should be bounded. Indeed, committing one model element should not end up with committing all PU, MU and RU. We argue that the number of PU, MU and RU should be proportional to the number of elements that belong to the containment tree of the committed

element. This number is always bounded with models that follow a power law, which is the case of UML models.

The second principle that governs scalability is the ratio of modified elements versus versioned elements. While only one element is committed, it represents all its containment tree. As a consequence, all the containment is committed. When those elements are committed, the version of their corresponding PU, MU and RU are increased. However, those PU, MU and RU certainly possess other elements. As a consequence, those elements are impacted by the commit although they have not been modified. We consider that the root model element and its containment tree are the truly modified elements (*ModE*). All the elements that belong to the corresponding PU, MU and RU are versioned elements (*VersE*). Regarding scalability, we argue that *ModE* should be close to *VersE (ideally ModE = VersE).*

7. REFERENCES

[1] Blanc, X., Mougenot, A., Mounier, I. and Mens. T. Incremental detection of model inconsistencies based on model operations. CAiSE'09. 21st Conference on Advanced Informatin Systems Engineering. Amsterdam, The Netherlands. 2009.

[2] Object Management Group. *The Meta-Object Facility Core*. www.omg.org/mof/

[3] Object Management Group. The Object Constraint Language, Version 2.2. http://www.omg.org/spec/OCL/2.2/

[4] Object Management Group. Meta-Object Facility (MOF2.0) Query/View/Transformation 1.0 http://www.omg.org/spec/QVT.

[5] Object Management Group. *The XML Metadata Interchange.* www.omg.org/technology/documents/formal/xmi.htm.

[6] O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly. 2009.

[7] Mougenot, A., Blanc, X. and Gervais, M.P. *D-Praxis: a peer-to-peer collaborative editing framework.* DAIS'09. 9th Internation Conference on Distributed Applicatio] n and Interoperable Systems.

[8] Sriplakich, P., Blanc, X. and Gervais, M.P. *Collaborative software engineering on large-scale models: requirements and experience in ModelBus*. SAC'08. ACM Symposium on Applied Computing. Fortaleza, Ceara, Brazil. 2008.

[9] Sprilakich, P. *ModelBus: un environment réparti et ouvert pour l'ingéniérie de modèles.* PhD. Thesis. Université Pierre et Marie Curie, Paris, France, 2007. (In spite of its French title, the thesis is written in English).

Appendix 1. IOCL specification of the *diff* operations

IOCL is part of the OMG MOF-QVT (Meta-Object Facility Query, View, Transform) standard [4] for model transformation specification. Its original purpose is to support procedural specifications of imperatively executable model transformations. It is a simple and intuitive extension of the OCL, (another OMG standard, part of UML2, to specify constraints on UML models and MOF meta-models [3]) with basic imperative constructs such a variable definition, variable assignment, conditionals and loops. The goal of the IOCL specifications is merely to prove that the diff and merge algorithm models provided in the present D2.2 deliverable are sufficiently precise to serve as a sound basis for implementation. It is in no way prescriptive. Once the algorithm has been understood, a given implementation might provide the specified services in alternative fashions using alternative data structures (for example to satisfy specific non-functional requirements or reuse legacy tools).

```
1  context GalaxyQueryProcessor::diffModelElt(projId: String, meId: String, localParticId: String, remoteParticId: String): MeDiff
2  body: do { var localCu:CollabUnit := getCollabUnit(projId, localParticId);
              -- get the local collaborative unit localCu for project whose id is projId and whose participant is localParticId

3              var localRev:Integer = localCu->head->tip.revision
              -- get revision number of last commit in the local head branch of localCu

4              var localMe:ModelElt := getModelElt(projId, meID, localParticId, localRev);
              -- get latest local version localMe of modelElt whose Id is meId in localCu

5              var baseMe:ModelElt := getModelElt(projId, meID, localparticId, localRev-1);
              -- get base version baseMe of modelElt whose Id is meId in local Cu

6              var diffBase2LocalActions:ModelAction[*] := localMe->mda;
              -- the action sequence from baseMe to localMe was stored in the galaxy when the same participant last called commitl
or update on localMe;

7              var remoteCu:CollabUnit := getCollabUnit(projId, remoteParticId);
              -- get the remote collaborative unit remoteCu for project whose id is projId and whose participant is remoteParticId

8              var remoteRev:Integer = remoteCu->head->tip.revision
              -- get revision number of last commit in the remote head branch of remoteCu

9              var remoteMe:ModelElt := getModelElt(projId, meID, remoteParticId, remoteRev);
              -- get latest remote version remoteMe of modelElt whose Id is meId in remoteCu
```

```
10          compute diffBase2RemoteActions:ModelAction[*] := remoteMe.mda
11           {
12              var prevRemote : ModelElt := remoteMe->prevRev;
13              var prevRemoteRev:Integer = prevRev.revision;
14              while (prevRemoteRev => localRev-1)
15               {
16                 prevRemote := prevRemote->prevRev;
17                 prevRemoteRev := prevRemoteRev -1;
18                 diffBase2RemoteActions := diffBase2RemoteActions->prepend(prevRemote.mda);
19               }
20           }
            -- the action sequence from the BaseMe to RemoteMe is constructed by iterating over prevRev links from remoteMe
            -- until reaching a model element which revision number matches that of baseMe;

21          var inLocalNotRemote:ModelAction[*] := diffBase2LocalActions - diffBase2RemoteActions;
            -- the actions in the baseMe to localMe sequence, but not in baseMe to remoteMe sequence;

22          var inRemoteNotLocal:ModelAction[*] := diffBase2RemoteActions - diffBase2LocalActions;
            -- the actions in the baseMe to remoteMe sequence, but not in the baseMe to localMe;

23          diffLocalRemote := new MeDiff;
            -- create new MeDiff object;

24          diffLocalRemote.local := localMe;
25          diffLocalRemote.remote := remoteMe;
26          diffLocalRemote.base := baseMe;
            -- fills it local (resp. remote, base) role with localMe (resp. remoteMe, baseMe);

27          diffLocalRemote.base2local := diffBase2LocalActions;
28          diffLocalRemote.base2remote := diffBase2RemoteActions;
            -- fills its base2local (resp. base2remote) role with the actions sequence from baseMe to localMe (resp. remoteMe)

29          diffLocalRemote.outMrAddedToLocalNotRemote := inLocalNotRemote->select(ar:AddRef | ar.from = localMe)-
>collect(result);
            -- the references added from baseMe to localMe but not to remoteMe are obtained by selecting the outgoing AddRef
actions
            -- from the sequence from baseMe to localMe minus the sequence from baseMe to remoteMe

30          diffLocalRemote.outMrAddedToRemoteNotLocal := inRemoteNotLocal->select(ar:AddRef | ar.from = remoteMe)-
>collect(result);
            -- the references added from baseMe to remoteMe but not to localMe are obtained by selecting the outgoing AddRef
actions
            -- from the sequence from baseMe to remoteMe minus the sequence from baseMe to localMe

31          diffLocalRemote.outMrDeletedToLocalNotRemote := inLocalNotRemote->select(ar:RmRef | ar.from = localMe)-
>collect(result);
            -- the references deleted from baseMe to localMe but not to remoteMe are obtained by selecting the outgoing RmRef
actions
            -- from the sequence from baseMe to localMe minus the sequence from baseMe to remoteMe
```

```
32          diffLocalRemote.outMrDeletedToRemoteNotLocal := inRemoteNotLocal-->select(ar:RmRef | ar.from = remoteMe)-
>collect(result);
            -- the references deleted from baseMe to remoteMe but not to localMe are obtained by selecting the outgoing RmRef
actions
            -- from the sequence from baseMe to remoteMe minus the sequence from baseMe to localMe

33          diffLocalRemote.inMrAddedToLocalNotRemote := inLocalNotRemote->select(ar:AddRef | ar.to = localMe)-
>collect(result);
            -- the references added from baseMe to localMe but not to remoteMe are obtained by selecting the ingoing AddRef
actions
            -- from the sequence from baseMe to localMe minus the sequence from baseMe to remoteMe

34          diffLocalRemote.inMrAddedToRemoteNotLocal := inRemoteNotLocal->select(ar:AddRef | ar.to = remoteMe)-
>collect(result);
            -- the references added from baseMe to remoteMe but not to localMe are obtained by selecting the ingoing AddRef
actions
            -- from the sequence from baseMe to remoteMe minus the sequence from baseMe to localMe

35          diffLocalRemote.inMrDeletedToLocalNotRemote := inLocalNotRemote->select(ar:RmRef | ar.to = localMe)-
>collect(result);
            -- the references deleted from baseMe to localMe but not to remoteMe are obtained by selecting the ingoing RmRef
actions
            -- from the sequence from baseMe to localMe minus the sequence from baseMe to remoteMe

36          diffLocalRemote.inMrDeletedToRemoteNotLocal := inRemoteNotLocal-->select(ar:RmRef | ar.to = remoteMe)-
>collect(result);
            -- the references deleted from baseMe to remoteMe but not to localMe are obtained by selecting the ingoing RmRef
actions
            -- from the sequence from baseMe to remoteMe minus the sequence from baseMe to localMe

           diffLocalRemote.outMrAddedToBothWithDiffTargets := localMe->sourceOf->iterate(r:ModelRef; av:AddVal |
localme.mda->
37          var finalAddValsInLocal:AddVal[*] := localMe.attr->iterate(a:Attribute; sv:AddVal | localMe.mda->select(sv: AddVal |
sv.host = localMe and sv.attr = a)->last());
            -- extract from the action sequence from baseMe to localMe the last AddVal action for each attribute

38          var finalAddValsInRemote:AddVal[*] := remoteMe.attr->iterate(a:Attribute; sv:AddVal | remoteMe.mda->select(sv:
AddVal | sv.host = remoteMe and sv.attr = a)->last());
            -- extract from the action sequence from baseMe to remoteMe the last AddVal action for each attribute

39          finalAddValsInLocal->asSet().symmetricDifference(finalAddValsInRemote->asSet())-> forEach(sv);
           -- for each action sv in the symmetric difference between the respective last AddVal action sequences from baseMe to
localMe and from baseMe to remoteMe, recasted as sets
40          {
41            mad := new MaDiff;
              -- create a new object mad of type MaDiff

42            mad.attr := sv.attr;
              -- concerning the attribute argument of sv and showing:

43            mad.valLocal := sv.val;
              -- the attribute value in localMe
```

```
44              mad.valRemote := localMe.attr->select(name = sv.attr.name)->val;
                -- the value of the same attribute in remoteMe

45              mad.valBase := mad.valRemote;
                -- and the value of the same attribute in baseMe

46              diffLocalRemote->maDiff := diffLocalRemote->maDiff->including(mad)
                -- add this new maDiff object to the maDiff role of the meDiff object under construction
47            }
48          var containedMeRus := localMe.contained->ru;
            -- get the reused units containing all the model elements with nesting reference ingoing to localMe or outgoing from
localMe;

49          var changedRefRus := containedMeRus->union(containedMeRus->mrFromTo->select(locallyChanged));
            -- get all the reuse units that contain model elements which reference from or to model elements in containedMeRus has
locally changed;

50          var changedPus := changedRefRus->pu->select(locallyChanged);
            -- get all the locally changed product units that contains reuse units in changedRefRus

51          changedPus->ru->me->forEach(me)
52            { diffLocalRemote.ramification := diffRemote.ramification->union(diffModelElt(projId, me.uuid, localParticId,
remoteParticId) }
            -- fill the ramification role of diffLocalRemote with the result of recursive calls to diffModelElts on each model element in
the reuse units contained in the product units changedPus

53          return diffLocalRemote;
            -- finally, return the meDiff object constructed by the preceding sequence of operation calls
54        }
```