- **MovePu**, moves a nested product unit to a new container product unit; reifies the operation *move(afId: String, ...)* of *CompositeArtifact* in Figure 1. when *afId* is the *uuid* of a *ProductUni*t object;
- **AddRu,** adds a reuse unit in a container product unit; reifies the operation *add(afId: String)* of *CompositeArtifact* in Figure 1. when *afId* is the *uuid* of a *ReuseUni*t object;
- **DelRu,** deletes a reuse unit from a container product unit; reifies the operation *del(afId: String)* of *CompositeArtifact* in Figure 1. when *afId* is the *uuid* of a *ReuseUni*t object
- **MoveRu**, moves a reuse unit to a new container product unit; reifies the operation *move(afId: String, ...)* of *CompositeArtifact* in Figure 1. when *afId* is the *uuid* of a *ReuseUni*t object;
- **AddMu,** adds a method unit to a container product unit; reifies the operation *add(afId: String)* of *CompositeArtifact* in Figure 1. when *afId* is the *uuid* of a *MethodUni*t object
- **DelMu,** deletes an method unit from a container product unit; reifies the operation *del(afId: String)* of *CompositeArtifact* in Figure 1. when *afId* is the *uuid* of a *Method*bject
- **MoveMu**, moves a method unit to a new container product unit; reifies the operation *move(afId: String, ...)* of *CompositeArtifact* in Figure 1. when *afId* is the *uuid* of a *MethodUni*t object;
- **AddMe,** adds a model element to a reuse unit; reifies the operation *add(me: ModelElt)* of *ReuseUnit;*
- **DelMe,** deletes a model element from a reuse unit; reifies the operation *del(me: ModelElt)* of *ReuseUnit*
- **MoveMe,** moves a model element to a new reuse unit; reifies the operation *move(me: ModelElt, to: ResueUnit)* of *ReuseUnit;*
- **AddMr,** adds a model reference to a reuse unit; reifies the operation *add(mr: ModelRef)* of *ReuseUnit;*
- **DelMr,** deletes a model reference from a reuse unit; ; reifies the operation *add(mr: ModelRef)* of *ReuseUnit*
- **MoveMr,** moves a model reference to a new reuse unit; reifies the operation *move(mr: ModelRef, to: ResueUnit)* of *ReuseUnit;*
- **AddMa,** adds an attribute to a model element; reifies the operation *addMa(ma: Attribute)* of *ModelElt;*
- **DelMa,** deletes an attribute from a model element; reifies the operation *delMa(ma: Attribute)* of *ModelElt;*
- **MoveMa,** moves a model attribute to another model element; reifies the operation *move(ma: Attribute, to: ModelElt)* of *ModelEl*t;
- **AddMav,** adds a value to an attribute; reifies the operation *addMav(mav: PrimitiveValueSpec)* of *Attribute;*
- **RmMav,** removes a value from an attribute; reifies the operation *rmMav(mav: PrimitiveValueSpec)* of *Attribute;*

– **ShowMe,** adds a model element to a view; reifies the operation *showMe(me: ModelElt)* of *View*;
– **HideMe:** hides a model element in a view; reifies the operation*HideMe(me: ModelElt)* of *View*;
– **ShowMa,** adds a model element attribute to a view; reifies the operation *showMa(ma: Attribute)* of *View*
– **HideMa:** hides a model element attribute in aview; reifies the operation *HideMa(ma: Attribute)* of *View*

As shown in Figure 8, one operation of a model fragmentation strategy defines a mapping from a sequence of Revision Unit actions (*i.e.,* actions on model elements and views received from a CASE tool by a galaxy framework instance) to a sequence of artifact actions (on artifacts and the model elements and views they contain). This design allows to simultaneously: (a) represent Commit and Diff objects as action sequences, which was shown scalable in distributed modeling environments with trivial revision policy such as Praxis [4], [15] , and (b) decouple the representation of model elements and views for user-friendly modeling edition purposes from their representation for scalable collaborative revision control. In turn, this decoupling allows to experiment with a variety of such artifact grains to achieve revision control scalability, in a way that is transparent to the CASE tools connected to the galaxy and, consequently, their users. This is important since different revision policies may be adapted only to projects within a certain range of model size, team size, project lifetime etc. Therefore, the revision policy may have to evolve during the course of the project. Decoupling it from the model element and view representation of the CASE tools circumvents the need for such representation every time the revision policy is changed. Details on the interfaces between CASE tools using the galaxy and the galaxy framework that realize this decoupling are given in the next section.

6. CONTROLLING THE COLLABORATIVE UNIT LIFECYCLE FROM AN MDE CASE TOOL

In this section, we explain how the concept of collaborative unit can be leveraged to support transparent interoperability among potentially heterogeneous CASE tools used by collaborative MDE software projects.

6.1 APPROACH: DECOUPLING AND SEPARATION OF CONCERNS

In the previous section, we explained how the distinction between edit actions performed in an MDE CASE tool and artifact actions performed by a collaborative unit helps decoupling the former from the later. In Figure 7, we propose five separate interfaces to connect an MDE CASE tool to an instance of the galaxy framework. This proposal constitutes a possible initial blueprint for tasks T4.1 (architecture specification) and T4.2 (galaxy core framework) of the project that depend on D2.1. It illustrates how the concept of collaborative unit defined in the present deliverable can be leveraged for these tasks. The key idea of this proposal is to distinguish between five largely separate concerns of revision control and provide one interface and corresponding realization class (or component) for each such concern. These interfaces are described in turn in each of the following subsections.

6.2 THE GALAXY QUERY API

The API *GalaxyQuery* provides read-only operations allowing an MDE CASE tool to query the collaborative units of a galaxy framework instantiation. It includes operations to search for galaxy users, projects, project participants, collaborative units, push and pull collaboration relations between remote collaborative units, development history tags, including branches and locks, model elements, references and views. It also includes diff operations to compare model elements and views and an audit operation to detect and return inconsistencies in model elements and views. Finally, it includes the *checkOut* operation that returns the model element or view found at the tip of the current branch in the local collaborative unit associated with a given project-user pair.

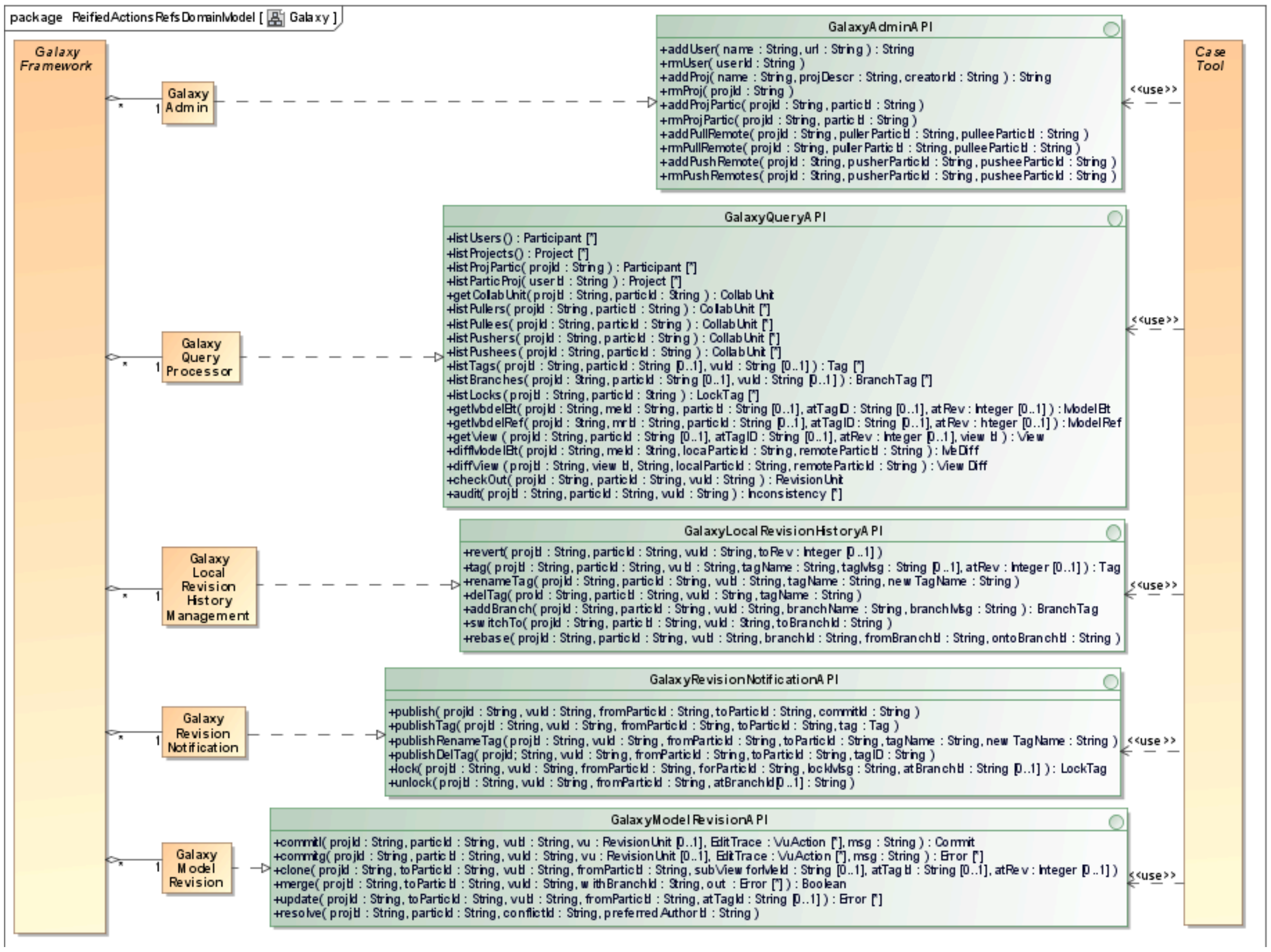

package ReifiedActionsRefsDomainModel [Galaxy]
Galaxy Framework
Case Tool
Galaxy Admin
Galaxy Query Processor
Galaxy Local Revision History Management
Galaxy Revision Notification
Galaxy Model Revision
*
1
<<use>>
GalaxyAdminAPI
+addUser(name : String, url : String) : String
+rmUser(userId : String)
+addProj(name : String, projDescr : String, creatorId : String) : String
+rmProj(projId : String)
+addProjPartic(projId : String, particId : String)
+rmProjPartic(projId : String, particId : String)
+addPullRemote(projId : String, pullerParticId : String, pulleeParticId : String)
+rmPullRemote(projId : String, pullerParticId : String, pulleeParticId : String)
+addPushRemote(projId : String, pusherParticId : String, pusheeParticId : String)
+rmPushRemotes(projId : String, pusherParticId : String, pusheeParticId : String)
GalaxyQueryAPI
+listUsers() : Participant [*]
+listProjects() : Project [*]
+listProjPartic(projId : String) : Participant [*]
+listParticProj(userId : String) : Project [*]
+getCollabUnit(projId : String, particId : String) : CollabUnit
+listPullers(projId : String, particId : String) : CollabUnit [*]
+listPullees(projId : String, particId : String) : CollabUnit [*]
+listPushers(projId : String, particId : String) : CollabUnit [*]
+listPushees(projId : String, particId : String) : CollabUnit [*]
+listTags(projId : String, particId : String [0..1], vuId : String [0..1]) : Tag [*]
+listBranches(projId : String, particId : String [0..1], vuId : String [0..1]) : BranchTag [*]
+listLocks(projId : String, particId : String) : LockTag [*]
+getModelElt(projId : String, meId : String, particId : String [0..1], atTagID : String [0..1], atRev : Integer [0..1]) : ModelElt
+getModelRef(projId : String, mrId : String, particId : String [0..1], atTagID : String [0..1], atRev : Integer [0..1]) : ModelRef
+getView (projId : String, particId : String [0..1], atTagID : String [0..1], atRev : Integer [0..1], viewId) : View
+diffModelElt(projId : String, meId : String, locaParticId : String, remoteParticId : String) : MeDiff
+diffView (projId : String, viewId, String, localParticId : String, remoteParticId : String) : View Diff
+checkOut(projId : String, particId : String, vuId : String) : RevisionUnit
+audit(projId : String, particId : String, vuId : String) : Inconsistency [*]
GalaxyLocalRevisionHistoryAPI
+revert(projId : String, particId : String, vuId : String, toRev : Integer [0..1])
+tag(projId : String, particId : String, vuId : String, tagName : String, tagMsg : String [0..1], atRev : Integer [0..1]) : Tag
+renameTag(projId : String, particId : String, vuId : String, tagName : String, newTagName : String)
+delTag(projId : String, particId : String, vuId : String, tagName : String)
+addBranch(projId : String, particId : String, vuId : String, branchName : String, branchMsg : String) : BranchTag
+switchTo(projId : String, particId : String, vuId : String, toBranchId : String)
+rebase(projId : String, particId : String, vuId : String, branchId : String, fromBranchId : String, ontoBranchId : String)
GalaxyRevisionNotificationAPI
+publish(projId : String, vuId : String, fromParticId : String, toParticId : String, commitId : String)
+publishTag(projId : String, vuId : String, fromParticId : String, toParticId : String, tag : Tag)
+publishRenameTag(projId : String, vuId : String, fromParticId : String, toParticId : String, tagName : String, newTagName : String)
+publishDelTag(projId; String, vuId : String, fromParticId : String, toParticId : String, tagID : String)
+lock(projId : String, vuId : String, fromParticId : String, forParticId : String, lockMsg : String, atBranchId : String [0..1]) : LockTag
+unlock(projId : String, vuId : String, fromParticId : String, atBranchId[0..1] : String)
GalaxyModelRevisionAPI
+commitl(projId : String, particId : String, vuId : String, vu : RevisionUnit [0..1], EditTrace : VuAction [*], msg : String) : Commit
+commitg(projId : String, particId : String, vuId : String, vu : RevisionUnit [0..1], EditTrace : VuAction [*], msg : String) : Error [*]
+clone(projId : String, toParticId : String, vuId : String, fromParticId : String, subViewforMeId : String [0..1], atTagId : String [0..1], atRev : Integer [0..1])
+merge(projId : String, toParticId : String, vuId : String, withBranchId : String, out : Error [*]) : Boolean
+update(projId : String, toParticId : String, vuId : String, fromParticId : String, atTagId : String [0..1]) : Error [*]
+resolve(projId : String, particId : String, conflictId : String, preferredAuthorId : String)

Figure 7: Interfaces between the galaxy framework and MDE CASE tools

This model element or view is passed to the CASE tool to provide the context for the next Revision Unit revision sequence executed by its user. The basic operation cycle executed by such a participant through a CASE tool connected to the galaxy thus consists of the three step sequence *checkOut, revise, commit*. Optionally, a *lock* can be additionally executed before the *revise*, with an accompanying *unlock* after the *commit*. There is no revise operation in galaxy framework API, since model and view revision actions are executed only by the CASE tool on its owned internal model representation. The revise action sequence is then passed to the galaxy framework instance as a parameter to the *commit* operations of the API.

6.3 THE GALAXY ADMIN API

The API *GalaxyAdmin* provides operations to administrate the galaxy. It includes operation to add and delete users and projects to and from the galaxy, to add and delete users to and from the participant list of a project, and to establish and close push and pull collaboration relations between remote collaborative units. The network formed by these collaboration relationships defines the collaborative workflow for the project.

6.4 THE GALAXY LOCAL REVISION HISTORY API

The API GalaxyLocalRevisionHistory provides operations to create, delete and navigate among tag objects of a collaborative units. The collaborative unit design presented in section **Erreur ! Source du renvoi introuvable.** and inspired from the data structures used by state-of-the-art DRCS for code-driven software engineering such as git, hg and bzr allows decoupling these operations from the ones that manipulate the artifact objects that contain model element and view versions.

6.5 THE GALAXY REVISION NOTIFICATION

The GalaxyRevisionNotification API provides operations allowing a collaborative unit Cu to notify remote collaborative units, which are registered as puller of Cu, that stable new revisions are available at Cu. It includes operations to notify the creation of new commit objects (and consequently of the new artifact versions to which this object points) and new tag objects, including branch and lock tags. It also includes the operation unlock used to notify release of a lock tag.

6.6 THE GALAXY MODEL REVISION API

The GalaxyModelRevision API provides operations to exchange model elements and views among a CASE tool, the local collaborative unit to which it is connected and the remote collaborative units from which to pull updated versions of these elements and views that were concurrently committed there by local collaborative units of other project participants.

These operations include:

- **cloning** part of a remote collaborative unit into the local collaborative to initiate collaboration on a model element or view;
- **committing** from the CASE tool a new version of a model element or view unit, either to the local collaborative (decentralized workflow, using the *commitl* operation), or to a single blessed reference remote collaborative unit (automated centralized workflow, using the *commitg* operation);
- **merging** two development branches available at the local collaborative unit;
- **updating** the local collaborative unit with a new version of a model element or view pulled from a remote collaborative unit; this sometimes involves performing first a merge between two change sets executed concurrently by two different participants on their respective CASE tools;
- **resolving** the conflicts that can result from a merge attempt.

6.7 THE GALAXY CLASSES REALIZING THE GALAXY APIS

One possible simple architecture blueprint for the galaxy framework, shown in Figure 7 and Figure 8 and, is to include one class (or component) to realize each of the five APIs described in the previous section. Although not shown in Figure 7 for conciseness's reason, each such class would possess one operation for each operation in the API that it realizes. These class operations would realize the services offered by the API operations by calling the general operations defined in the collaborative unit definition of section 3

Many such calls are fairly straightforward since the operations sets offered by the proposed galaxy framework API on the one hand, and defined in the classes defining the collaborative units on the other hand, are very similar. The main recurring differences between their respective signatures are the following. The first results from the fact that there is a distinct collaborative unit for each project-participant pair, whereas a CASE tool may be used by several users to participate to several projects

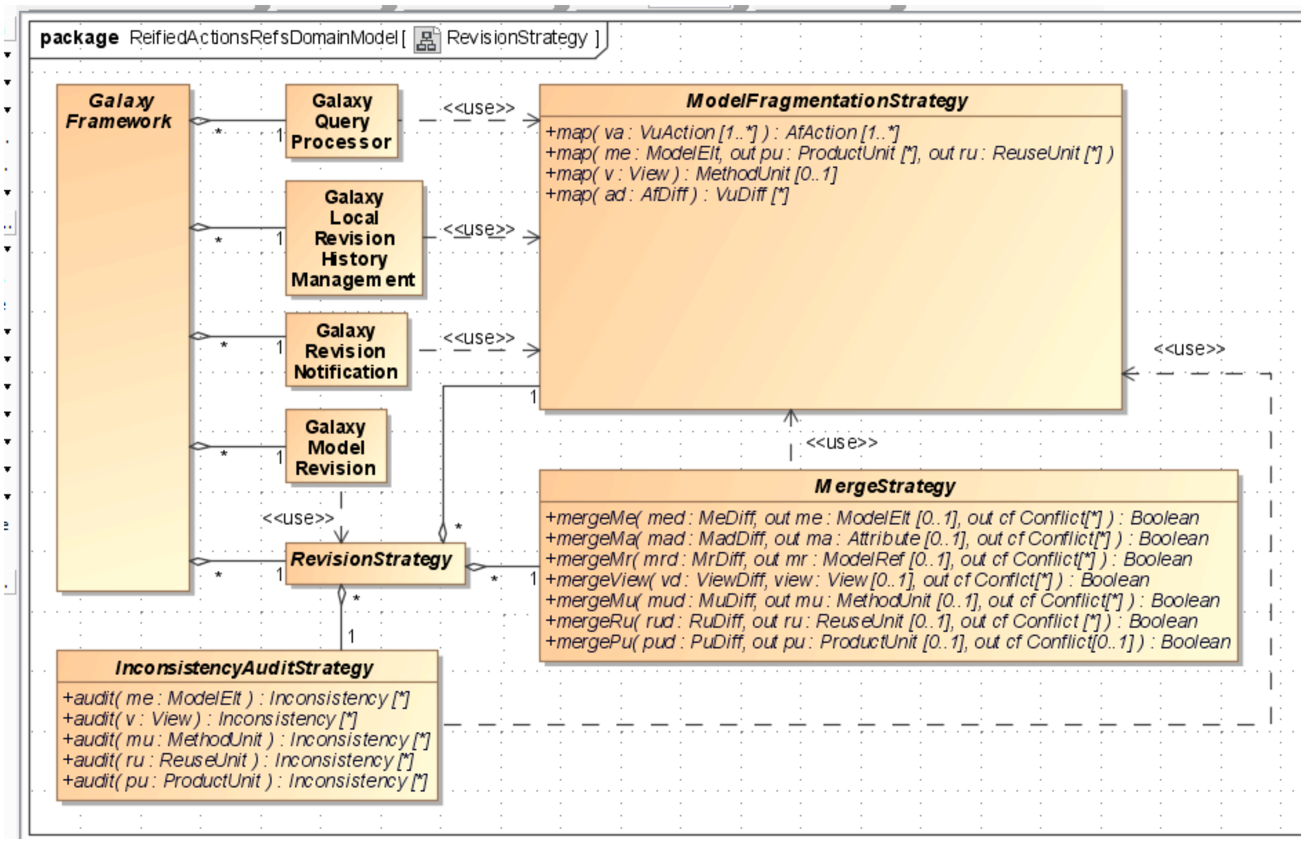

package ReifiedActionsRefsDomainModel [RevisionStrategy]
Galaxy Framework
Galaxy Query Processor
Galaxy Local Revision History Management
Galaxy Revision Notification
Galaxy Model Revision
<<use>>
ModelFragmentationStrategy
+map(va : VuAction [1..*]) : AfAction [1..*]
+map(me : ModelElt, out pu : ProductUnit [*], out ru : ReuseUnit [*])
+map(v : View) : MethodUnit [0..1]
+map(ad : AfDiff) : VuDiff [*]
MergeStrategy
+mergeMe(med : MeDiff, out me : ModelElt [0..1], out cf Conflict[*]) : Boolean
+mergeMa(mad : MadDiff, out ma : Attribute [0..1], out cf Conflict[*]) : Boolean
+mergeMr(mrd : MrDiff, out mr : ModelRef [0..1], out cf Conflict[*]) : Boolean
+mergeView(vd : ViewDiff, view : View [0..1], out cf Conflict[*]) : Boolean
+mergeMu(mud : MuDiff, out mu : MethodUnit [0..1], out cf Conflict[*]) : Boolean
+mergeRu(rud : RuDiff, out ru : ReuseUnit [0..1], out cf Conflict [*]) : Boolean
+mergePu(pud : PuDiff, out pu : ProductUnit [0..1], out cf Conflict[0..1]) : Boolean
RevisionStrategy
InconsistencyAuditStrategy
+audit(me : ModelElt) : Inconsistency [*]
+audit(v : View) : Inconsistency [*]
+audit(mu : MethodUnit) : Inconsistency [*]
+audit(ru : ReuseUnit) : Inconsistency [*]
+audit(pu : ProductUnit) : Inconsistency [*]
*
1

Figure 8 : Revision strategy

Therefore, the API operations include two additional parameters to identify the project concerned by the operation call and the participant executing it. The second difference between galaxy framework API operation signatures and the collaborative unit operation signatures results from the decoupling in the galaxy framework between Revision Units (i.e., model elements and views) on the one hand, and the artifacts (i.e., product, reuse and method units) used to persistently store them for revision control. Therefore, while the API operations include Revision Unit parameter identifiers, the collaborative units include artifact identifiers. The classes realizing the galaxy framework API must therefore maps the former into the latter, by using the relationships modeled as associations between the two shown in Figure 4. The third differences between the respective signatures of operations in the galaxy framework API and collaborative unit definitions is that where the former includes version unit action sequences the latter includes artifact action sequences.

As shown in Figure 8, the classes realizing API operations that contain parameter which type is an action sequence or control meta-data represented using such sequences depends on the mapping from Revision Unit action sequences to artifact action sequence encapsulated into a the specific model fragmentation strategy chosen for a particular project. Figure 8 illustrate the key idea of the proposed abstract galaxy framework: to be configurable through the specialization of three abstract classes that serve as intervening processing layer between the operations of concrete classes realization the galaxy framework API and the corresponding operations of the collaborative units and their components.

These three classes are

- *ModelFragmentationStrategy*, which define how to group model elements and views in product, reuse and method units;
- *MergeStrategy* which defines, whenever possible, how to automatically merge pairs of differing product units, reuse units, method units and the model elements and views that they persistently store;
- *InconsistencyAuditStrategy,* which defines what constitute an automatically detectable inconsistency in a model element, a model view, method unit, reuse unit and product unit.

A concrete revision strategy consists of an assembly of one concrete specialization of each of these three classes. Figure 8 also shows which of the classes realizing the galaxy framework API depends (i.e., uses the operations) depends on which of these three strategy components.

7. AN EXAMPLE OF COLLABORATIVE UNIT LIFECYCLE

The previous section showed a high-level blueprint for a galaxy framework that leverages the concept of collaborative unit defined in the preceding sections to support collaborative model construction. The present section gives a simple illustrative example script for such construction using the framework outlined in the previous section. This script involves three participants respectively called yb, jr and xb. The collaborative workflow pattern instantiated in this script is the single blessed repository managed by a human gatekeeper. The gatekeeper is responsible for creating the project in the galaxy, registering its participants, setting up the appropriate collaborative relationships between them, and guaranteeing the consistency of the blessed repository. In the script, galaxy user yb is the gatekeeper, while the two other participants jr and xb are mere developers.

7.1 CONFIGURING THE COLLABORATIVE WORKFLOW AMONG GALAXY USERS

Before leveraging the galaxy framework to construct a model, a preliminary step is required. It involves registering users to the framework, creating a project, assigning users to the project and defining the workflow that these users must follow to collaborate on the project.

The preliminary step of our illustrative example script is shown in Figure 9 in the form of a sequence diagram with eight lifelines:

1. ybCt representing the CASE tool of the gatekeeper yb;
2. jrCt representing the CASE tool of developer jr;
3. xbCt representing the CASE tool of developer xb;
4. GalaxyAdmin representing the GalaxyAdmin API
5. pj1 representing the collaborative MDE project;
6. yb representing the project participant and gatekeeper yb;
7. jr representing the project participant jr;
8. xb representing the project participant xb;

The diagram features 17 messages between these lifelines that illustrate the detail usage of the GalaxyAdmin API to initiate a project and set up a given collaborative workflow. Messages number 1-9 create the project, register the three galaxy users and declare them as collaborators of the project. Messages number 10-17 define the collaborative workflow for the project.

They allow the gatekeeper yb to push and pull revisions to and form the collaborative units of both jr and xb. Conversley, both jr and xb can push and pull revisions to and from the collaborative unit of yb. However, jr and xb cannot connect to each others' collaborative unit. They thus only collaborate indirectly through the mediation of the gatekeeper yb. This is an example of distributed revision control workflow that still retains some centralized communication aspects

Having presented the preliminary configuration step of our illustrative script, we can now proceed with presenting its model construction and revision steps. This is the object of the next eight subsections. Each such section follows the execution of one revision step in the example script. It contains a stereotyped class diagram that shows a content snapshot for the three collaborative units involved in this three participant script. In these diagrams we use UML packages for two distinct purposes. The first use is to represent the three collaborative units of this galaxy instance. By convention, each collaborative unit is called by adding the suffix Cu (Collaborative unit) after the name of the participant owning it. The second use of UML package is to represent UML model element of metaclass Package. By convention, packages representing collaborative units are white, while packages representing UML packages are blue.

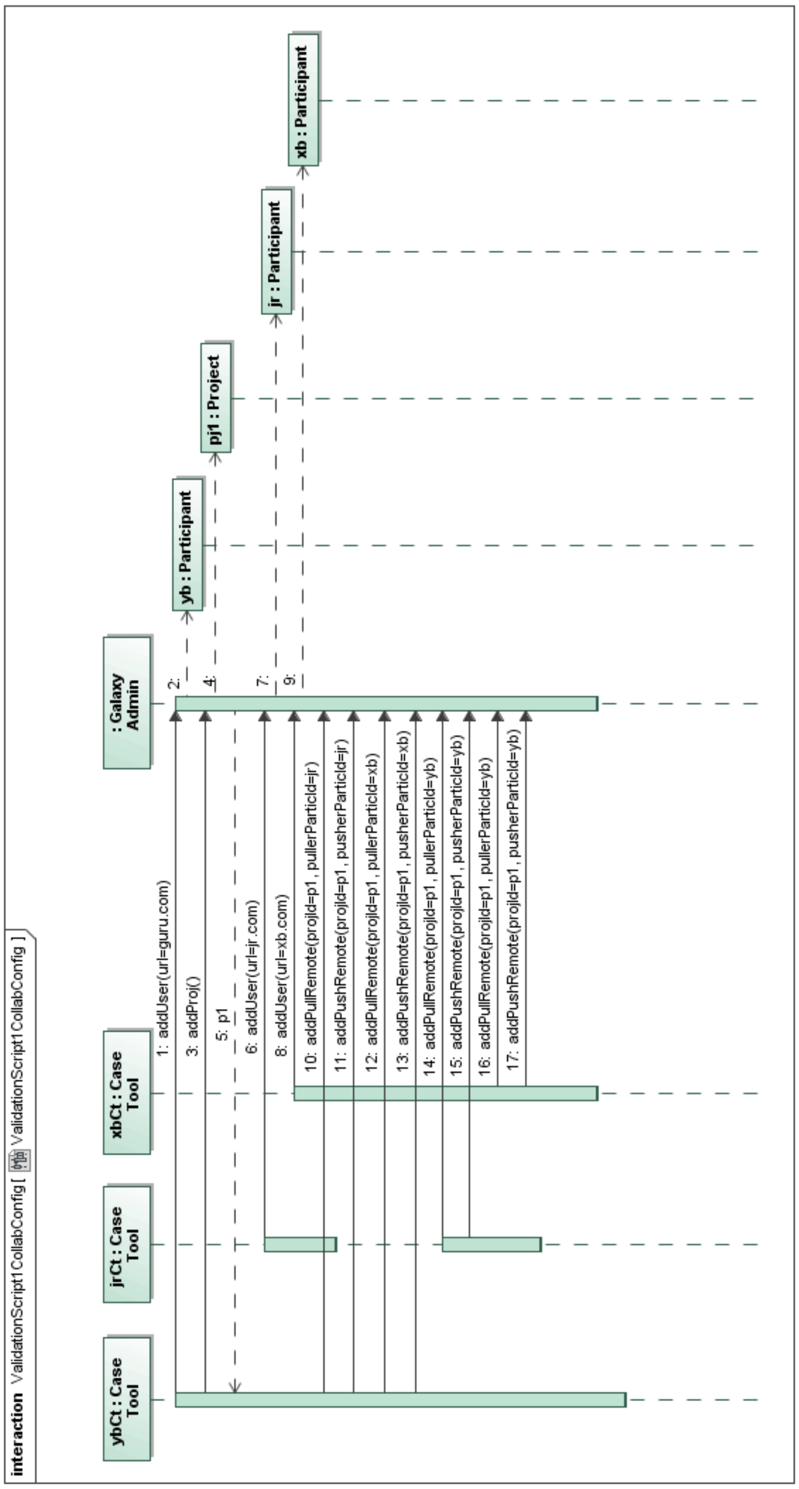

interaction ValidationScript1CollabConfig [ValidationScript1CollabConfig]
ybCt : Case Tool
jrCt : Case Tool
xbCt : Case Tool
: Galaxy Admin
yb : Participant
pj1 : Project
jr : Participant
xb : Participant
1: addUser(url=guru.com)
2:
3: addProj()
4:
5: p1
6: addUser(url=jr.com)
7:
8: addUser(url=xb.com)
9:
10: addPullRemote(projId=p1, pullerParticId=jr)
11: addPushRemote(projId=p1, pusherParticId=jr)
12: addPullRemote(projId=p1, pullerParticId=xb)
13: addPushRemote(projId=p1, pusherParticId=xb)
14: addPullRemote(projId=p1, pullerParticId=yb)
15: addPushRemote(projId=p1, pusherParticId=yb)
16: addPullRemote(projId=p1, pullerParticId=yb)
17: addPushRemote(projId=p1, pusherParticId=yb)

Figure 9: Setting up a collaboration workflow between three Galaxy users

In addition to this dual use of packages, another non-standard notations used in these snapshot diagrams are stereotypes of the form <<vN>> where N is an integer. When it decorates packages used to represent a collaborative unit, N represents the step in the script to which the collaborative unit state shown in the diagram corresponds. When it decorates a model element, it represents its revision number. A third non-standard notation used in these model revision script snapshots are dependency relationships between packages representing collaborative units. Each such dependency is stereotyped by the galaxy API operation call sequence that triggered information to be sent from collaborative unit source of the dependency to the collaborative unit target of the dependency. Operations call sequences are represented by calls between curly brackets and separated by semi-colons in both the dependency stereotypes and the notes associated to them.

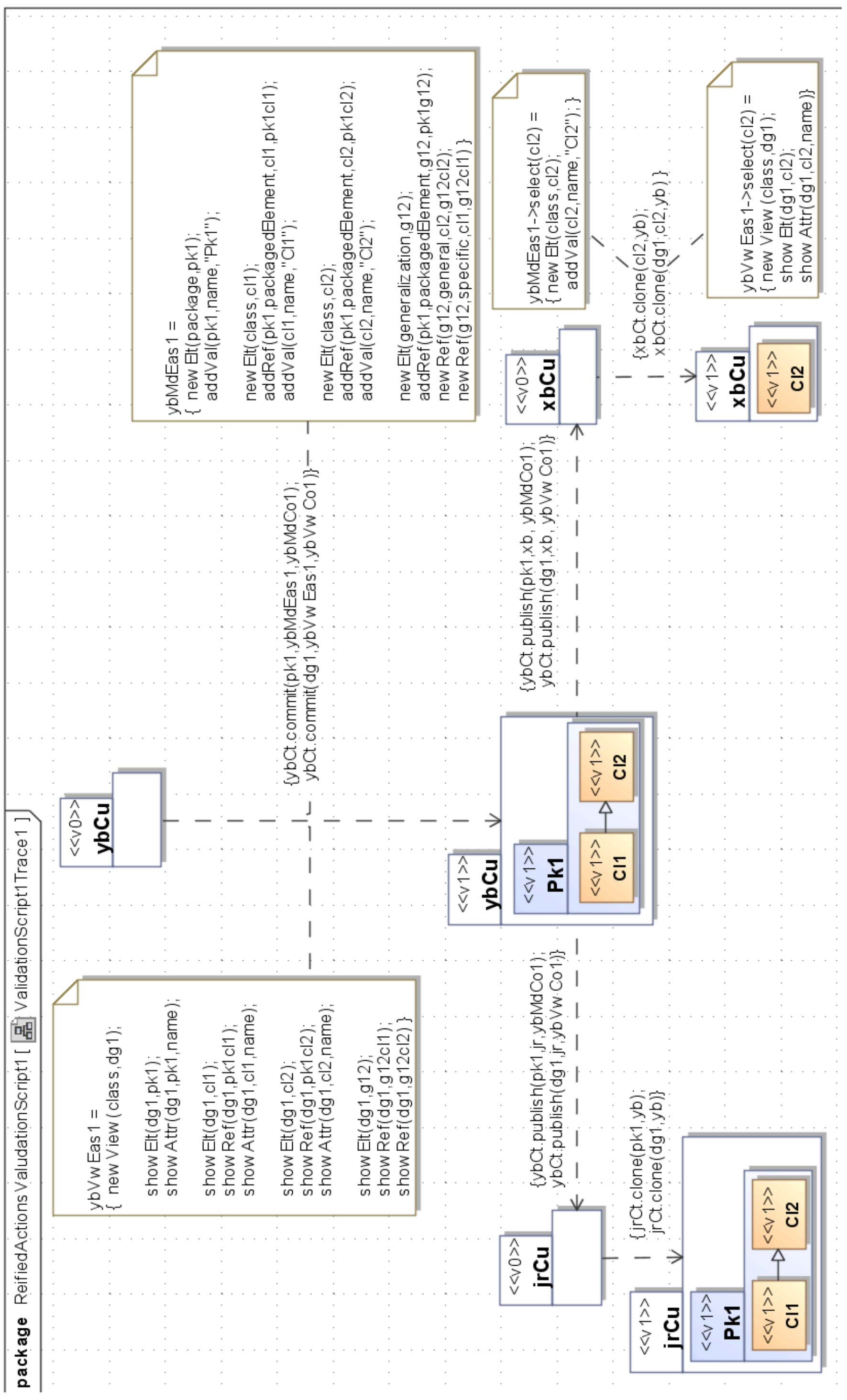
7.2 STEP 1: BLESSED REPOSITORY GATEKEEPER CREATES FIRST VERSION OF THE MODEL, PUBLISHES IT AND THE COLLABORATORS CLONE IT

During this step, shown in Figure 10, the gatekeeper participant yb uses his CASE tool to first create an initial version of the UML model to build collaboratively leveraging the galaxy. For this purpose, he calls the commit operation twice, first with a parameter instantiated by a model edition action sequence, ybMdEas1, and then with a parameter instantiated by a view action sequence, ybVwEas1.

The execution of these operations results in:

- A UML model in which a package Pk1 containing two classes, Cl1 and Cl2, the latter generalizing the former;
- A commit object ybMdCo1 that points the artifact containing the elements of the UML model.
- A UML class diagram showing all the model elements, references and attributes of the UML model;

A commit object ybVwCo1 that points to the arefact containing the diagram representation of the UML model.



```
ybMdEas1 =
{ new Elt(package,pk1);
  addVal(pk1,name,"Pk1");

  new Elt(class,cl1);
  addRef(pk1,packagedElement,cl1,pk1cl1);
  addVal(cl1,name,"Cl1");

  new Elt(class,cl2);
  addRef(pk1,packagedElement,cl2,pk1cl2);
  addVal(cl2,name,"Cl2");

  new Elt(generalization,g12);
  addRef(pk1,packagedElement,g12,pk1g12);
  new Ref(g12,general,cl2,g12cl2);
  new Ref(g12,specific,cl1,g12cl1) }
```

```
ybVwEas1 =
{ new View(class,dg1);

  showElt(dg1,pk1);
  showAttr(dg1,pk1,name);

  showElt(dg1,cl1);
  showRef(dg1,pk1cl1);
  showAttr(dg1,cl1,name);

  showElt(dg1,cl2);
  showRef(dg1,pk1cl2);
  showAttr(dg1,cl2,name);

  showElt(dg1,g12);
  showRef(dg1,g12cl1);
  showRef(dg1,g12cl2) }
```

```
{ybCt.commit(pk1,ybMdEas1,ybMdCo1);
 ybCt.commit(dg1,ybVwEas1,ybVwCo1)}
```

```
{ybCt.publish(pk1,xb, ybMdCo1);
 ybCt.publish(dg1,xb, ybVwCo1)}
```

```
ybMdEas1->select(cl2) =
{ new Elt(class,cl2);
  addVal(cl2,name,"Cl2"); }
```

```
{xbCt.clone(cl2,yb);
 xbCt.clone(dg1,cl2,yb) }
```

```
ybVwEas1->select(cl2) =
{ new View(class,dg1);
  showElt(dg1,cl2);
  showAttr(dg1,cl2,name)}
```

```
{ybCt.publish(pk1,jr,ybMdCo1);
 ybCt.publish(dg1,jr,ybVwCo1)}
```

```
{jrCt.clone(pk1,yb);
 jrCt.clone(dg1,yb)}
```

Figure 10: Blessed collaborative unit gatekeeper creates initial UML diagram and publishes it.

After having called these two commit operations, the gatekeeper yb calls the publish operation to notify the collaborative units of the two other participants, jr and xb, that a new model and a new view of it are now available. As parameters, these publish operations contains the commit objects that were created by the corresponding commit operations.

Upon receiving the notification, both jr and xb then concurrently call the clone operation to create copies of the model and diagram in their respective collaborative units. jr clones the model elements (and corresponding diagram) from the top-level element package Pk1. In contrast, xb clones only the model elements (and corresponding diagram) from the nested element Cl2. At the end of this step, both yb and jr then possess identical full models and views of the project's initial version in their respective collaborative units, whereas xb contains a partial copy of both with only Cl2.

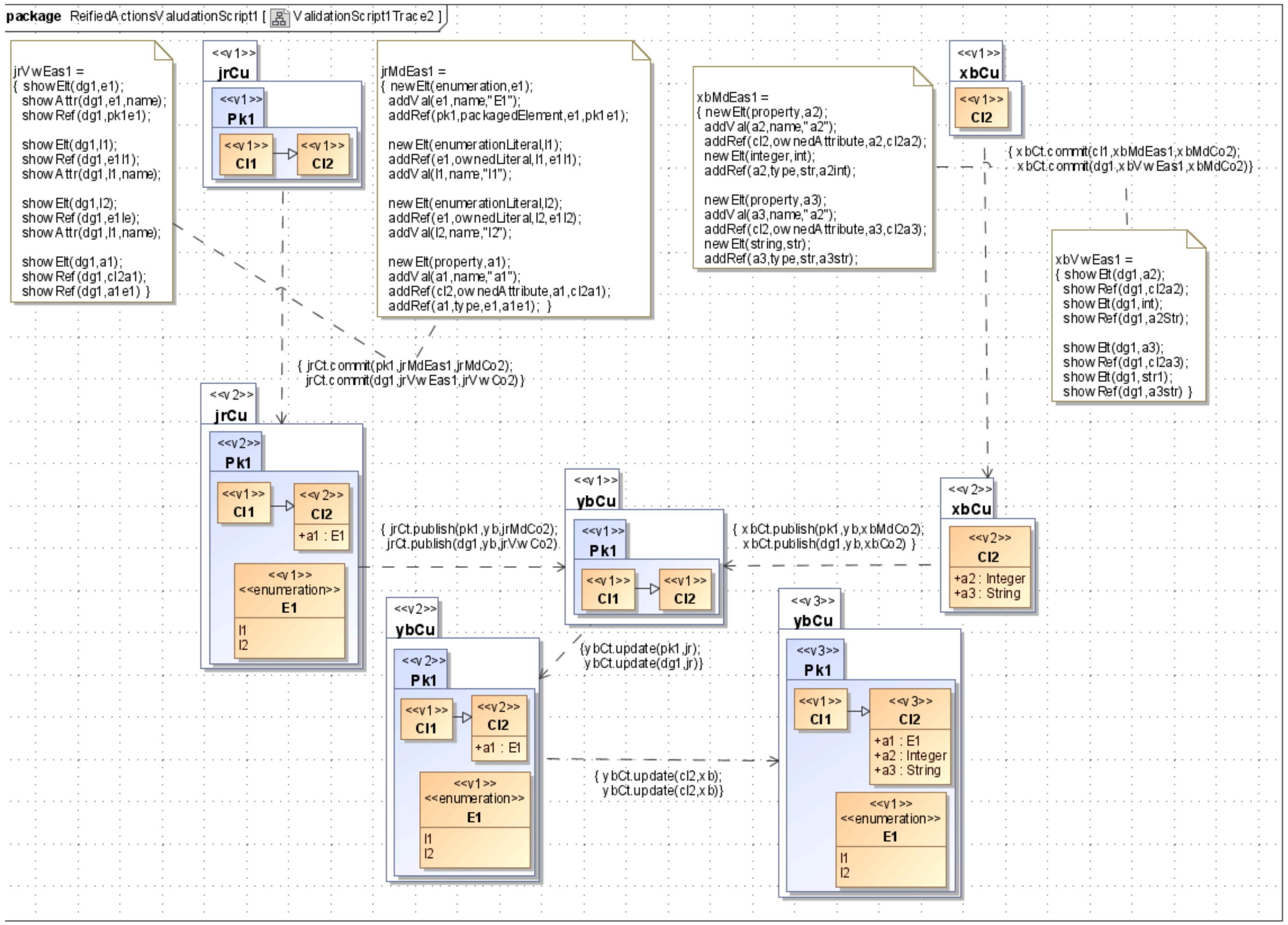
7.3 STEP 2: CONFLICT FREE CONCURRENT REVISIONS OF THE MODEL BY COLLABORATORS

The next step of the script is shown in Figure 11. This step starts by jr and xb concurrently making changes to their respective copies of project's first revision. They do so by calling the commit operation twice, one to persistently store in their respective collaborative units the change they made to the UML model elements and another one to persistently store in their respective collaborative units the changes they made to the UML diagram. This practice of making two calls to galaxy framework API operations, one to manipulate model elements, and another one to manipulate views and diagrams that show some of these elements, will be maintained throughout the presented script. We chose it to illustrate how the concerns of model revision (in our case the UML model element containment tree) and model view revision (in our case the UML class diagram displaying chosen elements in this tree) can be cleanly separated while using our proposed concepts of collaborative unit and galaxy framework API. However, it does not precludes a specific implementation of this framework to (a) force diagrams to be attached below some model element in the containment tree and (b) have a single call to the galaxy API operation to simultaneously execute on a model element and its associated diagrams. After locally committing their respective changes to their collaborative units, jr and xb then make them available to xb by (concurrently) calling publish operations. Note that this operation is merely a notification. It does not trigger any form of automatic update in the target collaborative unit.

Hence, after the execution of these publish operations, each of the three collaborative units of the galaxy contains a distinct UML model:

– yb's unit contains a diagram showing a package Pk1 containing a class Cl2 with no features generalizing a class Cl1 also without features;
– jr's unit contains a diagram showing a new revision of package Pk1 that contains a new enumeration E1 with two literals l1 and l2, in addition to Cl1, Cl2 and the generalization between the two that it shares with the revision in yb's collaborative unit; in jr's version, Cl2 also contain a new attribute a1 of type E1
– xb's unit contains a diagram showing only a new revision of the class Cl2 with two new attributes, a2 of type integer and a3 of type string.
– In all three units, the diagram provides a full disclosure of the UML model elements, references and attributes stored at that unit.

After receiving the notifications from both jr and xb, yb decides to first update his collaborative unit with the changes made by jr in his. The update operation calls a merge operation that take as parameter (a) the current base revision of the model element (or model view) stored in the local collaborative unit and (b) its more recent revision published by a remote collaborative unit. In our illustrative script, since the changes made by jr in his collaborative unit (remote for yb) were mere monotonic additions of new elements, the automatic merge (and thus the update that called it) of xb's collaborative unit with jr's succeeds and triggers no conflict. It results in xb's collaborative units being successfully updated with the content of jr's.



```
jrVwEas1 =
{ showElt(dg1,e1);
  showAttr(dg1,e1,name);
  showRef(dg1,pk1e1);

  showElt(dg1,l1);
  showRef(dg1,e1l1);
  showAttr(dg1,l1,name);

  showElt(dg1,l2);
  showRef(dg1,e1le);
  showAttr(dg1,l1,name);

  showElt(dg1,a1);
  showRef(dg1,cl2a1);
  showRef(dg1,a1e1) }
```

```
jrMdEas1 =
{ newElt(enumeration,e1);
  addVal(e1,name,"E1");
  addRef(pk1,packagedElement,e1,pk1e1);

  newElt(enumerationLiteral,l1);
  addRef(e1,ownedLiteral,l1,e1l1);
  addVal(l1,name,"l1");

  newElt(enumerationLiteral,l2);
  addRef(e1,ownedLiteral,l2,e1l2);
  addVal(l2,name,"l2");

  newElt(property,a1);
  addVal(a1,name,"a1");
  addRef(cl2,ownedAttribute,a1,cl2a1);
  addRef(a1,type,e1,a1e1); }
```

```
xbMdEas1 =
{ newElt(property,a2);
  addVal(a2,name,"a2");
  addRef(cl2,ownedAttribute,a2,cl2a2);
  newElt(integer,int);
  addRef(a2,type,str,a2int);

  newElt(property,a3);
  addVal(a3,name,"a2");
  addRef(cl2,ownedAttribute,a3,cl2a3);
  newElt(string,str);
  addRef(a3,type,str,a3str);
```

```
xbVwEas1 =
{ showElt(dg1,a2);
  showRef(dg1,cl2a2);
  showElt(dg1,int);
  showRef(dg1,a2Str);

  showElt(dg1,a3);
  showRef(dg1,cl2a3);
  showElt(dg1,str1);
  showRef(dg1,a3str) }
```

Figure 11: Conflict-free concurrent diagram revisions made by two collaborators

Then yb attempts to update his collaborative unit with the changes made by xb in his. Since these changes were also mere monotonic additions of new elements that were completely orthogonal to those concurrently made my jr's, this update also triggers no conflict and succeeds.

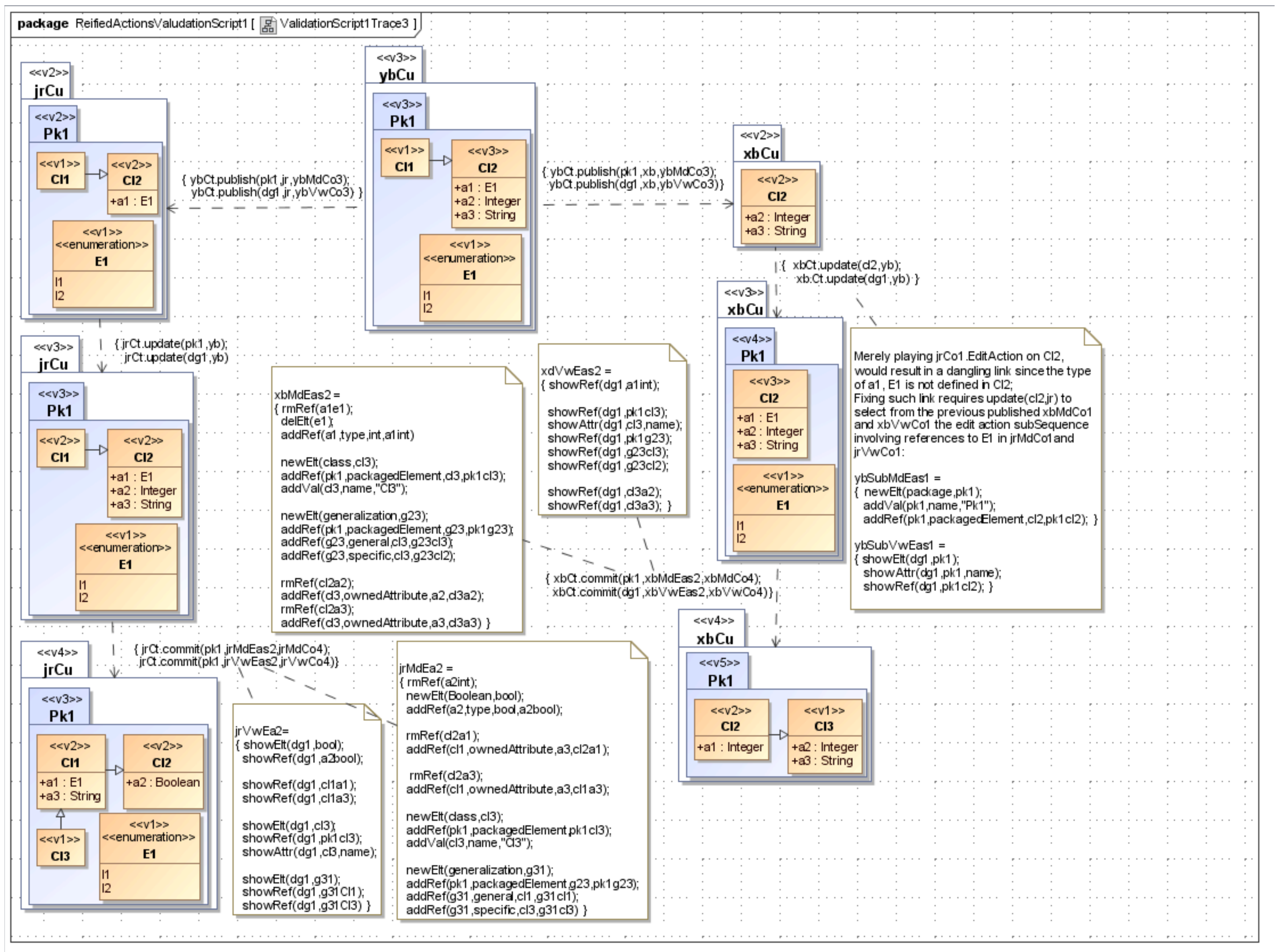
Step 3 continues at the top of Figure 12 that shows yb notifying both jr and xb that he just updated the content of his collaborative units with the content of theirs. He does so by calling the publish operation, once for the package pk1 model element, and then a second time for the diagram dg1. One important parameter of the first call is the commit objects xbMdCo3 that fills the tip role of the main branch of yb's collaborative units. For conciseness' sake, in Figure 12 and the rest of the script, we omit to show this main branch when it is the only one present in a collaborative unit. It is the third commit object for the pk1 model element in yb's collaborative unit. The first such commit object was created as a result of yb's initial commit. The second one resulted from the update operation that merged the result of this initial commit with the changes introduced and published by jr's first changes to package pk1. The third commit object resulted from the update operation that merged the result of this second commit with the changes introduced and published by xb's first changes to the class Cl2. Similarly, an important parameter of the second call is the commit object xbVwCo3. Just as xbMdCo3, it also fills the tip role of the main branch of yb's collaborative unit. But instead of pointed to the current third revision of root model element pk1, it points to the third revision of the diagram that visually displays this model element, together with its contained elements. Therefore, yb's collaborative units contains two parallel revision threads of commit objects on the main branch: one representing the history of artifacts persistently storing the model elements, and one representing the history of the model view provided by the class diagram.

7.4 STEPS 3 CONFLICTING CONCURRENT REVISIONS OF THE MODEL BY COLLABORATORS

At this point, jr and xb resume to work in parallel, each one updating their collaborative units with the changes published by yb. On jr's side the update simply results in incorporating the attributes a2 and a3, originally introduced by xb, into to the class Cl2.

On xb's side, the situation is more complex due to the different scope of the models stored in the respective collaborative units of yb and xb: the whole package Pk1 in yb's but only the class Cl2 in xb's. Since xb remains interested in working only on Cl2, considering that the other elements of Pk1 are irrelevant to his current modeling concern, he calls the update operation on Cl2 and not on Pk1. As a result, he gets the new attribute a1 for class Cl2. However, the type of a1 is the enumeration E1, which is defined outside of the scope of xb's current model.

This example points out to two possible alternatives for the detailed semantics of the update operation in this case. The first is to maintain the scope of xb's local copy of the model under construction to class Cl2 and warning him that the type of the newly added attribute a1 is outside of this scope, and therefore a dangling reference from a local perspective. The second possibility is to have update automatically extend the scope of xb's local model so as to avoid such local dangling references. In our particular case, this involves applying to xb's current model, the action sequences corresponding to the creation of Pk1, the creation of E1 and the creation of the nesting references from Pk1 to Cl2 and E1.



```
{ ybCt.publish(pk1,jr,ybMdCo3);
  ybCt.publish(dg1,jr,ybVwCo3) }

{ ybCt.publish(pk1,xb,ybMdCo3);
  ybCt.publish(dg1,xb,ybVwCo3)}

{ xbCt.update(cl2,yb);
  xb.Ct.update(dg1,yb) }

{ jrCt.update(pk1,yb);
  jrCt.update(dg1,yb)

{ xbCt.commit(pk1,xbMdEas2,xbMdCo4);
  xbCt.commit(dg1,xbVwEas2,xbVwCo4)}

{ jrCt.commit(pk1,jrMdEas2,jrMdCo4);
  jrCt.commit(pk1,jrVwEas2,jrVwCo4)}
```

```
xbMdEas2 =
{ rmRef(a1e1);
  delElt(e1);
  addRef(a1,type,int,a1int)

  newElt(class,cl3);
  addRef(pk1,packagedElement,cl3,pk1cl3);
  addVal(cl3,name,"Cl3");

  newElt(generalization,g23);
  addRef(pk1,packagedElement,g23,pk1g23);
  addRef(g23,general,cl3,g23cl3);
  addRef(g23,specific,cl3,g23cl2);

  rmRef(cl2a2);
  addRef(cl3,ownedAttribute,a2,cl3a2);
  rmRef(cl2a3);
  addRef(cl3,ownedAttribute,a3,cl3a3) }
```

```
xdVwEas2 =
{ showRef(dg1,a1int);

  showRef(dg1,pk1cl3);
  showAttr(dg1,cl3,name);
  showRef(dg1,pk1g23);
  showRef(dg1,g23cl3);
  showRef(dg1,g23cl2);

  showRef(dg1,cl3a2);
  showRef(dg1,cl3a3); }
```

Merely playing jrCo1.EditAction on Cl2, would result in a dangling link since the type of a1, E1 is not defined in Cl2;
Fixing such link requires update(cl2,jr) to select from the previous published xbMdCo1 and xbVwCo1 the edit action subSequence involving references to E1 in jrMdCo1and jrVwCo1:

```
ybSubMdEas1 =
{ newElt(package,pk1);
  addVal(pk1,name,"Pk1");
  addRef(pk1,packagedElement,cl2,pk1cl2); }

ybSubVwEas1 =
{ showElt(dg1,pk1);
  showAttr(dg1,pk1,name);
  showRef(dg1,pk1cl2); }
```

```
jrVwEa2=
{ showElt(dg1,bool);
  showRef(dg1,a2bool);

  showRef(dg1,cl1a1);
  showRef(dg1,cl1a3);

  showElt(dg1,cl3);
  showRef(dg1,pk1cl3);
  showAttr(dg1,cl3,name);

  showElt(dg1,g31);
  showRef(dg1,g31Cl1);
  showRef(dg1,g31Cl3) }
```

```
jrMdEa2 =
{ rmRef(a2int);
  newElt(Boolean,bool);
  addRef(a2,type,bool,a2bool);

  rmRef(cl2a1);
  addRef(cl1,ownedAttribute,a3,cl2a1);

  rmRef(cl2a3);
  addRef(cl1,ownedAttribute,a3,cl1a3);

  newElt(class,cl3);
  addRef(pk1,packagedElement,pk1cl3);
  addVal(cl3,name,"Cl3");

  newElt(generalization,g31);
  addRef(pk1,packagedElement,g23,pk1g23);
  addRef(g31,general,cl1,g31cl1);
  addRef(g31,specific,cl3,g31cl3) }
```

Figure 12: Conflicting concurrent diagram revisions made by two collaborators (part 1)

At this point, it is worth to notice several things:

The two publish operation calls that yb sent to xb's collaborative unit (respectively shown in Figure 10 and Figure 11) contained commit object parameters; recall from the collaborative unit definition in

Figure 1 that such object points to the action sequence used to insert the committed model and model view elements inside the artifact persistently storing them for revision purpose. If we assume that xb's collaborative units upon reception of the publish notification from yb's collaborative units, persistently stored the uuids of these remote commit objects, it can now request the sub-action sequences, remotely stored under these uuids in yb's collaborative unit, that its update operation now may need to minimally extend the scope of xb's partial copy of yb's model so as to resolve its dangling references.

– This scope extension is not total, since class Cl1 is still not included in xb's model; indeed, it is not yet relevant to his current focus which is Cl2, since there is no reference from Cl2 to Cl1; while Pk1 is also not directly relevant to xb's current concern, it is the common container of Cl2, E1 and the reference from Cl2 to E1; thus, the new scope of xb's model must include it if it is to possess a single top-level containing model element which is often required by CASE tools;

– Had xb chosen in the first step of the script to clone the entire model rooted at Pk1 to start working on the project, instead of cloning only the sub-model rooted at Cl2, he would have avoided the need to either accommodate himself with a locally dangling reference or to require update operations with built-in scope extension preventing dangling references.

This simple example illustrates the trade-offs involved in choosing between potentially partial collaborative unit cloning, commits and updates and only global ones. While the former possess on-demand information flavor that seems more scalable at first sight, the latter allows the collaborative unit operation implementation to be far more simple. All three major DRCS for code-driven development, git, hg, and bzr only allow cloning, committing and updating of an entire project in their current releases. Only the experimental subtrees library of git [8] allows partial cloning, committing and updating. However its usage has not yet been tested on large software projects. Concerning this issue of global vs. local clone, commit and update operations, it is worth remembering that in our proposal, the concept of revision strategy allows decoupling these operations as provided in the galaxy framework API from the corresponding ones as provided in the collaborative unit. While the API operations take Revision Units and Revision Unit actions as parameters, the corresponding collaborative unit operations take as parameters artifacts and artifact actions. This decoupling offers the possibility to explore the trade-off between various policies with respect with operation scope, without having to change the CASE tools client of galaxy framework instantiations.

In the rest of this script, we assume that galaxy framework instantiation in use does support partial cloning, commits and updates. We also assume that the merge operation called by the update operation implements an automated minimal scope extension of partial models, which both prevents local dangling references and insures the presence of a single root model element containing all the others.

With these assumptions, the result of xb updating its collaborative unit from the changes published by yb at the top of Figure 12 is that xb's collaborative unit is now rooted at package Pk1, which contains the class Cl2 with attribute a1 originally introduced by jr and attributes a2 and a3 originally introduced by xb, as well as the enumeration E1 with literals l1 and l2 originally introduced by jr.

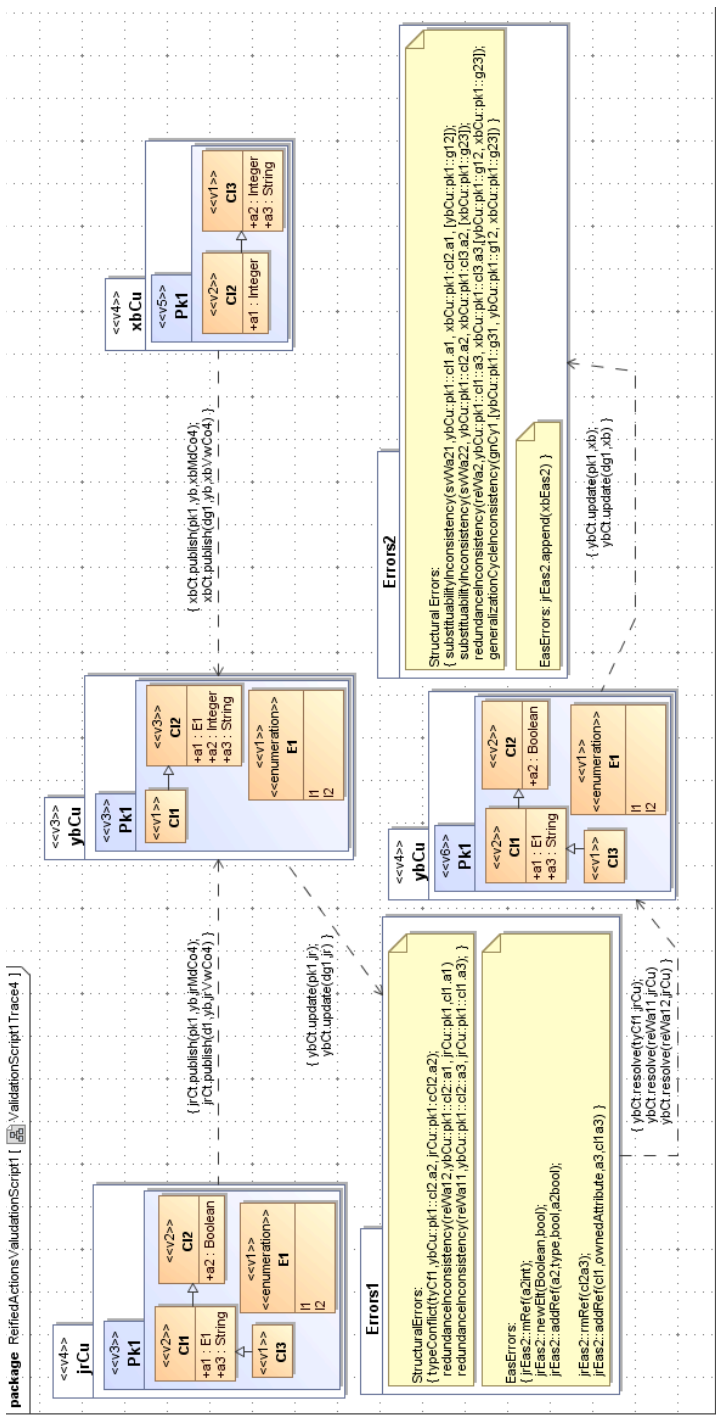
After this update and the concurrent update performed by jr, both jr and xb then concurrently execute concurrent commit operations. With these operations, jr changes the type of a2 from integer to Boolean, moves the a1 and a3 attributes form Cl2 to Cl1 and adds a new class Cl3 as a specialization of Cl1. On his side, xb changes the type of a1 from E1 to integer, deletes E1 which is no longer needed with this change (at least from the limited local perspective on which xb currently focuses), creates a new class Cl3 as a generalization of Cl2 and moves the attributes a2 and a3 from Cl2 to Cl3.

This step of the script continues at the top of

Figure 13, where both jr and xb concurrently publish these respective changes to the gatekeeper's collaborative unit. Like for the first set of publish notification that he had received from jr and xb, yb decides to first update his collaborative unit with content of jr's. This particular update execution results in one conflict and two inconsistencies shown in the package Errors1 of Figure 13.

The conflict concerns the type of the a2 attribute of class Cl2. While in yb's blessed collaborative unit this type is integer, in jr's collaborative unit this type Boolean. Since integer and Boolean are exclusive types in the UML, this type mismatch is a genuine semantic conflict. As a result, the respective copies of the attribute, the one in jr's collaborative unit and the one in yb's collaborative unit cannot be merged.

The two inconsistencies concern redundancies that would occur as a result of executing the merge: the attributes a1 of type E1 and a3 of type String would inherited by class Cl1 from its superclass Cl2 and redundantly defined with the same types in Cl1 itsef. Note, that there are two assumptions that need to be made about the galaxy framework instance in use for it to return these two inconsistencies as a result of the update operation call at the top left of Figure 14. The first is that this update operation automatically calls a merge operation that implements UML2's package merge specification. The second is that after the merge, the update operation then performs an audit operation which, among other things, performs redundancy analysis on the merge result.



```
StructuralErrors:
{ typeConflict(tyCf1,ybCu::pk1::cl2.a2, jrCu::pk1::cCl2.a2);
  redundanceInconsistency(reWa12,ybCu::pk1::cl2::a1, jrCu::pk1,cl1.a1)
  redundanceInconsistency(reWa11,ybCu::pk1::cl2::a3, jrCu::pk1::cl1.a3); }
```

```
EasErrors:
{ jrEas2::mRef(a2int);
  jrEas2::newElt(Boolean,bool);
  jrEas2::addRef(a2,type,bool,a2bool);

  jrEas2::rmRef(cl2a3);
  jrEas2::addRef(cl1,ownedAttribute,a3,cl1a3) }
```

```
Structural Errors:
{ substituabilityInconsistency(sWa21,ybCu::pk1::cl1.a1, xbCu::pk1::cl2.a1, [ybCu::pk1::g12]);
  substituabilityInconsistency(sWa22, ybCu::pk1::cl2.a2, xbCu::pk1::cl3.a2, [xbCu::pk1::g23]);
  redundanceInconsistency(reWa2,ybCu::pk1::cl1::a3, xbCu::pk1::cl3.a3,[ybCu::pk1::g12, xbCu::pk1::g23]);
  generalizationCycleInconsistency(gnCy1,[ybCu::pk1::g31, ybCu::pk1::g12, xbCu::pk1::g23]) }
```

```
EasErrors: jrEas2.append(xbEas2) }
```

Figure 13: Conflicting concurrent diagram revisions made by two collaborators (part 2)

This is a possible scenario, but not the only one. Alternative merge and audit strategies includes (a) performing the merge but not including the redundancy analysis in the audit and (b) performing a variation of the merge that automatically removes the redundancies by keeping attributes duplicated in several classes down a specialization path only at the highest level where they appear. Choosing among these alternatives is part of what we call the revision strategy. In the rest of our illustrative script, we will assume that every update first calls a UML package merge (or the specification of its recursive semantics on packageable element pairs for finer grain model elements such as classes) and then performs on the result an audit that detects redundancies.

The type conflict and the two attribute redundancy inconsistencies shown at the bottom left of Figure 13 are instances of the Error subclasses Conflict and MeBinInconsistency (respectively) defined in **Erreur ! Source du renvoi introuvable.**. Conflicts and such inconsistencies involve only two model elements. They are thus also instances of the class MeDiff to define in detail in the D2.2. However, we can anticipate that, as a difference specification, it will, just as AfDiff defined in Figure 1, have a role self, for the most recent local version, a role base for the preceding local version and a role with for an intervening remote version committed after base but before self[6]. The difference is that for a MeDiff these roles are filled with model elements, whereas for AfDiff they filled with artifacts. The conflict or inconsistency that such MeDiff introduce can thus be resolved simply by choosing for the updated revision either the model element filling the self role or the model element filling the with role. This is done by calling the operation resolve of the GalaxyModelRevisionAPI (see Figure 7). In the our validation script, after receiving the type conflict and redundancy inconsistencies as result of his attempt to update his collaborative units with the latest changes published by jr, yb resolves them by picking the with revision over his self revision in all three cases. As shown at the bottom center of Figure 14. These choices result in yb's collaborative unit containing exactly the same model and diagram as jr's.

[6] Svn calls *mine* the revision we call *self* and calls *other* the revision we call *with*.

Yb then attempts to update this result with the latest changes published by xb. This results in four inconsistencies shown at the bottom right of Figure 13. The first two are substitutability violations: Cl1 can no longer substitute (i.e., be used in operation invokations) its super class Cl2, since it locally redefines to enumeration E1 the type of the attribute a1 which it inherits with type integer from Cl12. Similarly, Cl2 can no longer substitute Cl3 since it locally redefines to Boolean the type of attribute a2 that it inherits from Cl3 with type integer. The third inconsistency is a redundancy warning signaling that attribute a3 locally defined in Cl1 is also inherited by Cl1 from Cl3 via Cl2.

These inconsistencies are cases of good design pattern violations. In contrast, the fourth and last one is a case of metamodel violation : the UML metamodel prohibits generalization cycles, but one would be introduced by merging package Pk1 from yb's collaborative unit with package Pk1 from xb's collaborative unit. Indeed, in the former Cl2 generalizes Cl1 which generalizes Cl3, while in the latter, Cl3 generalizes Cl2. The four generalizations resulting from merging the two would form a loop. All four inconsistencies are instances of the class MeNaryInconsistency defined in **Erreur ! Source du renvoi introuvable.**, since their detection involves more than two model elements not related by containment relationships. For example the fourth one involved six elements (three classes and three generalizations), that do not contain each other.

Sophisticated audit operations must be called by the merge operation (itself called by the update operation) in order to detect inconsistencies such as substitutability violations and generalization cycles. In addition to the model fragmentation strategy into product, reuse and method units and the merge strategy, a third, largely orthogonal aspects that compose a revision strategy is the audit strategy. It defines what errors in an automatic merge result must be automatically detected to trigger rollbacking the merge and require the modeler to change his version to resolve the inconsistency.

How yb deals with the inconsistencies just described is explained in the next step 5 and in Figure 13.

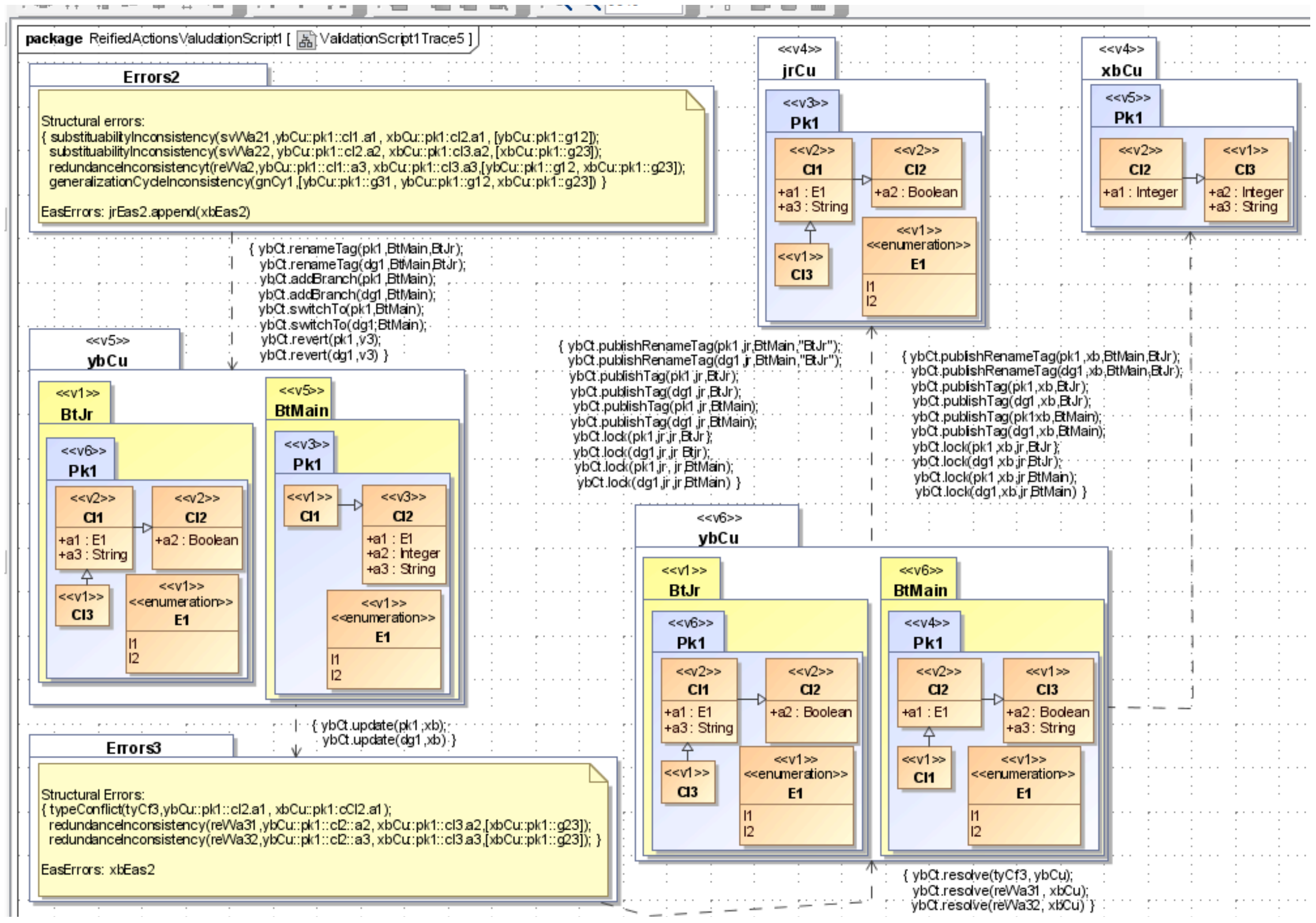
7.5 STEP 4 BLESSED COLLABORATIVE UNIT GATE KEEPER DELEGATES ERROR RESOLUTION BY CREATING BRANCHES AND PUBLISHING THEM

The gatekeeper yb realizes that the conflicts in the last updates that he respectively received from his two collaborators seem to stem from a disagreement among them. Since he has no strong personal preference between the two alternatives, he thus opts to create two development branches, one that merges yb's visions with jr's last revision, and another that merges yb's visions with xb's last revision. This is shown in Figure 14where we introduce an additional non-standard notation, using UML packages to distinguish distinct branches inside a collaborative unit. At this point we need to clarify that from the time they are created, each collaborative unit includes one main branch. We omitted the explicit representation of the main branch in the packages representing the collaborative units in the diagrams illustrating the first four steps of the script already presented only to avoid cluttering them.

Thus, the revision v4 of ybCu's collaborative unit should really contain a package called BtMain nested inside the package called ybCu and nesting the package called Pk1. Since yb had already solved the conflicts between his base revision and jr's last published one by choosing for all conflicts the alternative from jr's, before attempting the failed update with xb's last published changes, his current sole main branch contains the same model and diagram than the main branch in jr's collaborative unit. Yb thus starts by renaming his main branches BtJr, for both the artifacts storing the model elements and the artifacts storing the diagram displaying them. Since, as shown

in

Figure 1, in our proposed galaxy framework branches are special cases of tags, yb executes two calls to the operation renameTag. It then calls the operation addBranch twice, one to create a new branch for the model elements and another to create a new branch for the diagram. Note that in our proposed framework, creating a branch is merely creating a new BranchTag object that points to the tip Commit object for each artifact which revision history is stored in the collaborative unit.


package ReifiedActionsValudationScript1 [ValidationScript1Trace5]
Errors2
<<v5>> ybCu
<<v1>> BtJr
<<v6>> Pk1
<<v2>> Cl1 +a1 : E1 +a3 : String
<<v2>> Cl2 +a2 : Boolean
<<v1>> Cl3
<<v1>> <<enumeration>> E1 l1 l2
<<v5>> BtMain
<<v3>> Pk1
<<v1>> Cl1
<<v3>> Cl2 +a1 : E1 +a2 : Integer +a3 : String
<<v1>> <<enumeration>> E1 l1 l2
Errors3
<<v4>> jrCu
<<v3>> Pk1
<<v2>> Cl1 +a1 : E1 +a3 : String
<<v2>> Cl2 +a2 : Boolean
<<v1>> Cl3
<<v1>> <<enumeration>> E1 l1 l2
<<v4>> xbCu
<<v5>> Pk1
<<v2>> Cl2 +a1 : Integer
<<v1>> Cl3 +a2 : Integer +a3 : String
<<v6>> ybCu
<<v1>> BtJr
<<v6>> Pk1
<<v2>> Cl1 +a1 : E1 +a3: String
<<v2>> Cl2 +a2 : Boolean
<<v1>> Cl3
<<v1>> <<enumeration>> E1 l1 l2
<<v6>> BtMain
<<v4>> Pk1
<<v2>> Cl2 +a1 : E1
<<v1>> Cl3 +a2: Boolean +a3: String
<<v1>> Cl1
<<v1>> <<enumeration>> E1 l1 l2

```
Structural errors:
{ substituabilityInconsistency(svWa21 ,ybCu::pk1::cl1.a1 , xbCu::pk1::cl2.a1 , [ybCu::pk1::g12]);
  substituabilityInconsistency(svWa22, ybCu::pk1::cl2.a2, xbCu::pk1::cl3.a2, [xbCu::pk1::g23]);
  redundanceInconsistencyt(reWa2,ybCu::pk1::cl1::a3, xbCu::pk1::cl3.a3,[ybCu::pk1::g12, xbCu::pk1::g23]);
  generalizationCycleInconsistency(gnCy1 ,[ybCu::pk1::g31 , ybCu::pk1::g12, xbCu::pk1::g23]) }

EasErrors: jrEas2.append(xbEas2)
```

```
{ ybCt.renameTag(pk1 ,BtMain,BtJr);
  ybCt.renameTag(dg1 ,BtMain,BtJr);
  ybCt.addBranch(pk1 ,BtMain);
  ybCt.addBranch(dg1 ,BtMain);
  ybCt.switchTo(pk1,BtMain);
  ybCt.switchTo(dg1;BtMain);
  ybCt.revert(pk1 ,v3);
  ybCt.revert(dg1 ,v3) }
```

```
{ ybCt.publishRenameTag(pk1 ,jr,BtMain,"BtJr");
  ybCt.publishRenameTag(dg1 ,jr,BtMain,"BtJr");
  ybCt.publishTag(pk1 ,jr,BtJr);
  ybCt.publishTag(dg1 ,jr,BtJr);
  ybCt.publishTag(pk1 ,jr,BtMain);
  ybCt.publishTag(dg1 ,jr,BtMain);
  ybCt.lock(pk1 ,jr,jr,BtJr);
  ybCt.lock(dg1 ,jr,jr Btjr);
  ybCt.lock(pk1 ,jr, jr,BtMain);
  ybCt.lock(dg1 ,jr,jr,BtMain) }
```

```
{ ybCt.publishRenameTag(pk1 ,xb,BtMain,BtJr);
  ybCt.publishRenameTag(dg1 ,xb,BtMain,BtJr);
  ybCt.publishTag(pk1 ,xb,BtJr);
  ybCt.publishTag(dg1 ,xb,BtJr);
  ybCt.publishTag(pk1xb,BtMain);
  ybCt.publishTag(dg1 ,xb,BtMain);
  ybCt.lock(pk1 ,xb,jr,BtJr);
  ybCt.lock(dg1 ,xb,jr,BtJr);
  ybCt.lock(pk1 ,xb,jr,BtMain);
  ybCt.lock(dg1 ,xb,jr,BtMain) }
```

```
{ ybCt.update(pk1 ,xb);
  ybCt.update(dg1 ,xb) }
```

```
Structural Errors:
{ typeConflict(tyCf3,ybCu::pk1::cl2.a1 , xbCu::pk1::cCl2.a1 );
  redundanceInconsistency(reWa31 ,ybCu::pk1::cl2::a2, xbCu::pk1::cl3.a2,[xbCu::pk1::g23]);
  redundanceInconsistency(reWa32,ybCu::pk1::cl2::a3, xbCu::pk1::cl3.a3,[xbCu::pk1::g23]); }

EasErrors: xbEas2
```

```
{ ybCt.resolve(tyCf3, ybCu);
  ybCt.resolve(reWa31 , xbCu);
  ybCt.resolve(reWa32, xbCu) }
```

Figure 14: Blessed repository gate keeper delays conflict resolution by branching

This tip Commit object itself points to the most recent element of the revision history for the corresponding artifact.

Having renamed the old main branch BtJr and having created a new main branch BtMain had no effect on the current branch. After these two operations the main branch is still the old main branch which is now called BtJr. At this point yb wishes to incorporate the latest changes published by xb to the new main branch. To do so, he must first switch the current branch to this new branch BtMain.

Then, he must revert it to the last revision before the updates with jr's latest changes, since they introduce inconsistencies when merge with xb's lastest changes. He can now, as shown at the bottom left of Figure 13, update the new main branch with xb's lastest published changes7.

The attempt to update the reverted version of yb's main branch with xb's latest published changes fails, returning to the errors shown at the bottom left of Figure 13. The first is the conflict between the type of the attribute a1 of class Cl2: integer in xb's updated collaborative unit vs. enumeration E1 in yb's base collaborative unit. The second and third are warnings about the redundancy of having a2 and a3 attributes defined Cl2 in yb's base collaborative unit, while they are defined in Cl2's superclass Cl3 in xb's updated collaborative unit. Yb solves the type conflict by choosing a1's enumeration E1 type from his base collaborative unit, in effect rejecting xb's change of this type to integer. In contrast, he removes the warnings' causes by accepting xb's moving up a2 and a3 definition from Cl2 to its new superclass Cl3. The result of these calls to the resolve operation is the revision v6 of the branch BtMain in yb's collaborative unit.

Yb then publishes to both jr and xb the changes that occurred in his collaborative from revision v4 that was stored in his collaborative unit when he last received published changes from jr and xb, to the current revision v6. To do so he first notifies jr and xb of the changes he made on the

[7] Note that nearly all actions discussed in this script from now on is duplicated, one handling model elements and the other the model's diagram. In the rest of this explanation we will focus on describing the operations handling model elements since the one handling diagrams are identical except for the first parameter which is dg1 instead pk1.

branch structure of his collaborative units. He notifies his collaborators of the branch renaming between v4 and v6 by calling the operation publishRenameTag. He then notifies his collaborators of the updated content of this renamed branch by calling the operation publishTag. He then calls the same operation to notify his collaborators of the content of the newly created main branch. Finally, to avoid receiving again conflicting concurrent changes from his two collaborators, he locks the artifacts so that from then on only jr can now alter them. He notifies this locking to both jr and yb.

7.6 STEP 6 ONE COLLABORATOR MERGES DESIGN CHOICES FROM BOTH BRANCHES INTO ONE, DELETES THE OTHER AND PUBLISHES IT

With exclusive access to the galaxy model, jr then updates his collaborative units with the changes that yb just published. This operation and its result are shown on the left side of Figure 14. These changes not only include the changes made to the data in yb's collaborative units, i.e., the model elements and diagrams stored in the artifact objects, but also to changes made the development branching meta-data. The update thus takes into account the renaming of the main branch BtMain into BtJr, and the creation of a new branch called BtMain. Since yb created this BtJr branch to match the content of the latest version of jr's BtMain branch, the update his successful and jr's collaborative unit now contains an exact copy of xb's.

By executing a diff operation on the two branches, jr identifies their differences:

1. the location of class Cl3 in the class hierachy, below Cl1 in BtJr *vs.* above Cl2 in BtMain;
2. the location of attribute a1, in Cl1 in BtJr vs. in its superclass Cl2 in BtMain;
3. the location of attribute a2, in Cl2 in BtJr *vs.* in its super class Cl3 in BtMain;
4. the location of attribute a3, in Cl1 in BtJr *vs*. in its super-super class Cl3 in BtMain.

Each of these difference correspond to an error that yb received when attempting to update the last consistent revision of Pk1 from both jr's last published changes and xb's last published changes.

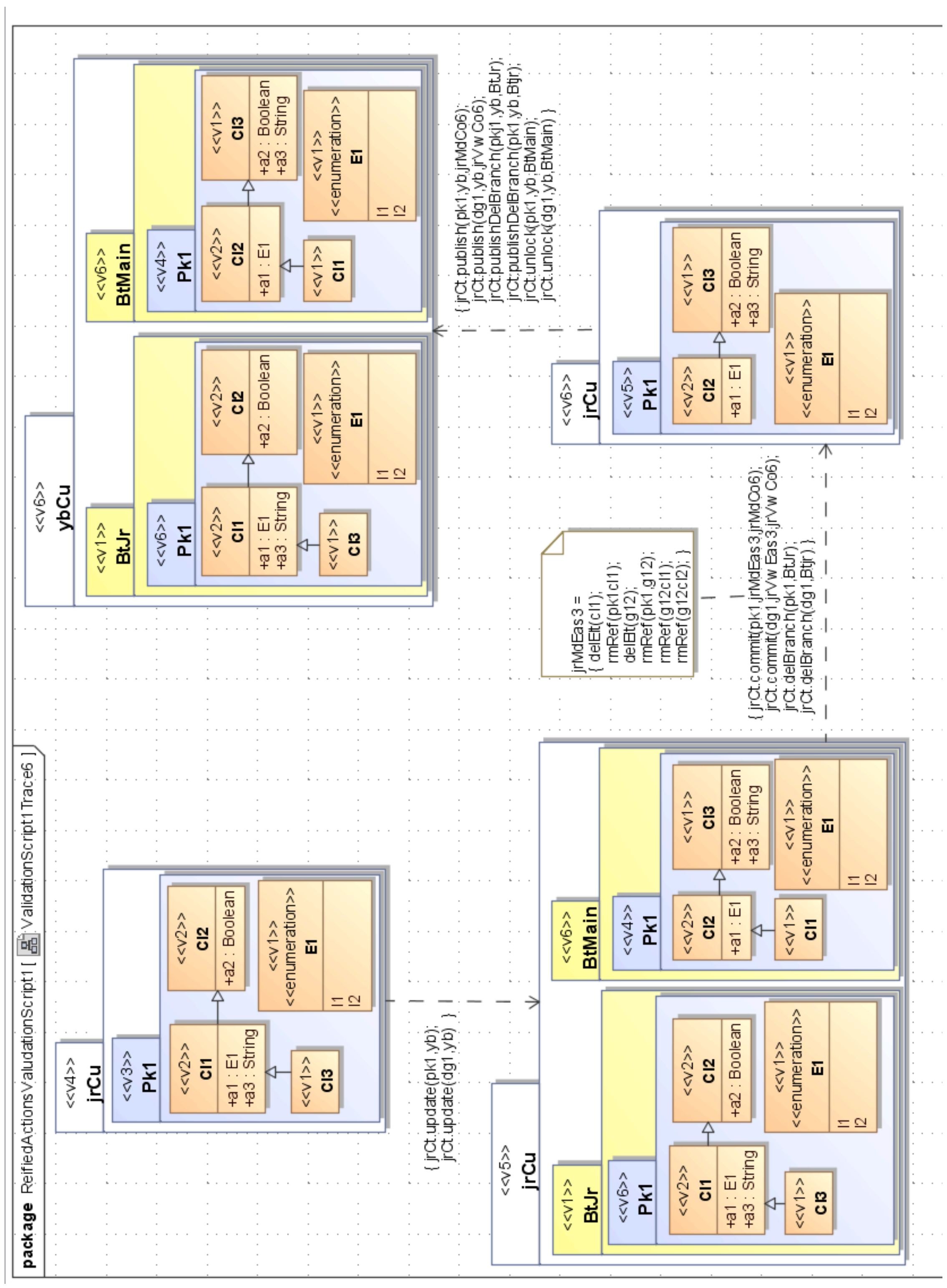

package ReifiedActionsValudationScript1 [ValidationScript1Trace6]
<<v6>> ybCu
<<v1>> BtJr
<<v6>> BtMain
<<v6>> Pk1
<<v4>> Pk1
<<v2>> Cl1
+a1 : E1
+a3 : String
<<v2>> Cl2
+a2 : Boolean
<<v1>> Cl3
+a2 : Boolean
+a3 : String
<<v1>> <<enumeration>> E1
l1
l2
<<v1>> Cl1
<<v1>> Cl3
{ jrCt.publish(pk1,yb,jrMdCo6); jrCt.publish(dg1,yb,jrVw Co6); jrCt.publishDelBranch(pkj1,yb,BtJr); jrCt.publishDelBranch(pk1,yb,Btjr); jrCt.unlock(pk1,yb,BtMain); jrCt.unlock(dg1,yb,BtMain) }
<<v6>> jrCu
<<v5>> Pk1
jrMdEas3 = { delElt(cl1); rmRef(pk1cl1); delElt(g12); rmRef(pk1,g12); rmRef(g12cl1); rmRef(g12cl2); }
{ jrCt.commit(pk1,jrMdEas3,jrMdCo6); jrCt.commit(dg1,jrVw Eas3,jrVw Co6); jrCt.delBranch(pk1,BtJr); jrCt.delBranch(dg1,Btjr) }
<<v4>> jrCu
<<v3>> Pk1
{ jrCt.update(pk1,yb); jrCt.update(dg1,yb) }
<<v5>> jrCu

Figure 15: One collaborator resolve conflicts and deletes useless branch

By locking Pk1 for exclusive access by jr, yb in effect delegated to jr the task of solving these inconsistencies. Jr does so by deleting Cl1 the model in the BtMain branch since it had no feature there and then deleting the BtJr branch.

These two operations correspond to choosing the second option for all four alternatives listed above. After executing the local change operations, jr publishes them to yb's collaborative unit and releases the lock.

As shown at the top of Figure 15, yb then uses those changes to updates his collaborative unit. In order to verify whether xb agrees with them, yb then publishes his updated collaborative unit to xb. He also locks it so that only xb can now change it.

7.7 STEP 7: COLLABORATOR AGREES WITH CHANGES MADE BY THE OTHER

Upon receive the latest changes from yb and the accompanying lock, xb now attempts to update his collaborative unit with these changes.

It results in two binary conflicts:

- The type of attribute a1, enumeration E1 in yb's latest revision *vs.* integer in xb's base revision;
- The type of attribute a2, Boolean in yb's latest revision *vs*. integer in xb's base revision.;

Xb resolves both these conflicts by choosing the types in yb's latest revision. This results in all three collaborators having the same model and diagram versions in their respective collaborative units. After resolving conflicts, xb's unlocks both the model and the diagram.

7.8 STEP 8 BLESSED COLLABORATIVE UNIT OWNER CREATES RELEASE1.0 I

In the last step of our galaxy framework illustrative use case, yb's tags the current revision of the model and diagram as release1.0 of the project. This is an appropriate moment, since at this point an identical and consistent copy of the project is stored in all its collaborative units. As shown in Figure 18, yb creates this release1.0 by first calling the operation *tag* on the model and diagram in his collaborative unit. He then calls the operation *publishTag* to notify his two collaborators, jr and xb, of this new tag. Since yb's last notification message to jr's collaborative unit was a lock on both the model and the diagram, xb additionally calls the operation unlock on both to release the lock and thus allows jr to resume making changes taking this first release as base revision.

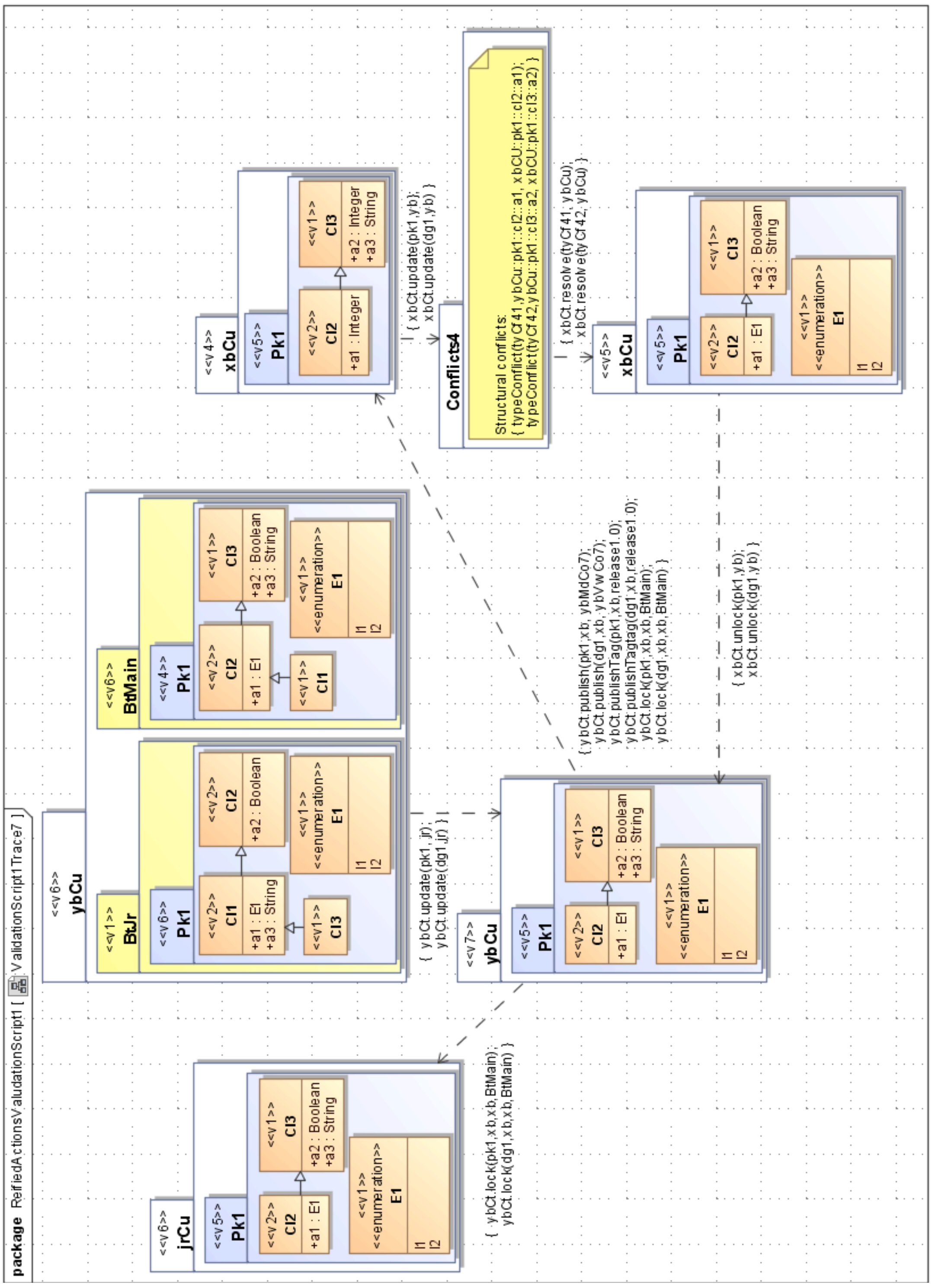

package ReifiedActionsValudationScript1 [ValidationScript1Trace7]
<<v6>> jrCu
<<v5>> Pk1
<<v2>> Cl2 +a1 : E1
<<v1>> Cl3 +a2 : Boolean +a3 : String
<<v1>> <<enumeration>> E1 l1 l2
<<v6>> ybCu
<<v1>> BtJr
<<v6>> Pk1
<<v2>> Cl1 +a1 : E1 +a3 : String
<<v2>> Cl2 +a2 : Boolean
<<v1>> Cl3
<<v1>> <<enumeration>> E1 l1 l2
<<v6>> BtMain
<<v4>> Pk1
<<v2>> Cl2 +a1 : E1
<<v1>> Cl3 +a2 : Boolean +a3 : String
<<v1>> Cl1
<<v1>> <<enumeration>> E1 l1 l2
<<v4>> xbCu
<<v5>> Pk1
<<v2>> Cl2 +a1 : Integer
<<v1>> Cl3 +a2 : Integer +a3 : String
{ xbCt.update(pk1,yb); xbCt.update(dg1,yb) }
Conflicts4
Structural conflicts: { typeConflict(tyCf41,ybCu::pk1::cl2::a1, xbCU::pk1::cl2::a1); typeConflict(tyCf42,ybCu::pk1::cl3::a2, xbCU::pk1::cl3::a2) }
{ xbCt.resolve(tyCf41; ybCu); xbCt.resolve(tyCf42, ybCu) }
<<v5>> xbCu
<<v5>> Pk1
<<v2>> Cl2 +a1 : E1
<<v1>> Cl3 +a2 : Boolean +a3 : String
<<v1>> <<enumeration>> E1 l1 l2
{ ybCt.update(pk1, jr); ybCt.update(dg1,jr) }
<<v7>> ybCu
<<v5>> Pk1
<<v2>> Cl2 +a1 : E1
<<v1>> Cl3 +a2 : Boolean +a3 : String
<<v1>> <<enumeration>> E1 l1 l2
{ ybCt.lock(pk1,xb,xb,BtMain); ybCt.lock(dg1,xb,xb,BtMain) }
{ ybCt.publish(pk1,xb, ybMdCo7); ybCt.publish(dg1,xb, ybVwCo7); ybCt.publishTag(pk1,xb,release1.0); ybCt.publishTagtag(dg1,xb,release1.0); ybCt.lock(pk1,xb,xb,BtMain); ybCt.lock(dg1,xb,xb,BtMain) }
{ xbCt.unlock(pk1,yb); xbCt.unlock(dg1,yb) }

Figure 16 : Collaborator agrees with changes made by the other.

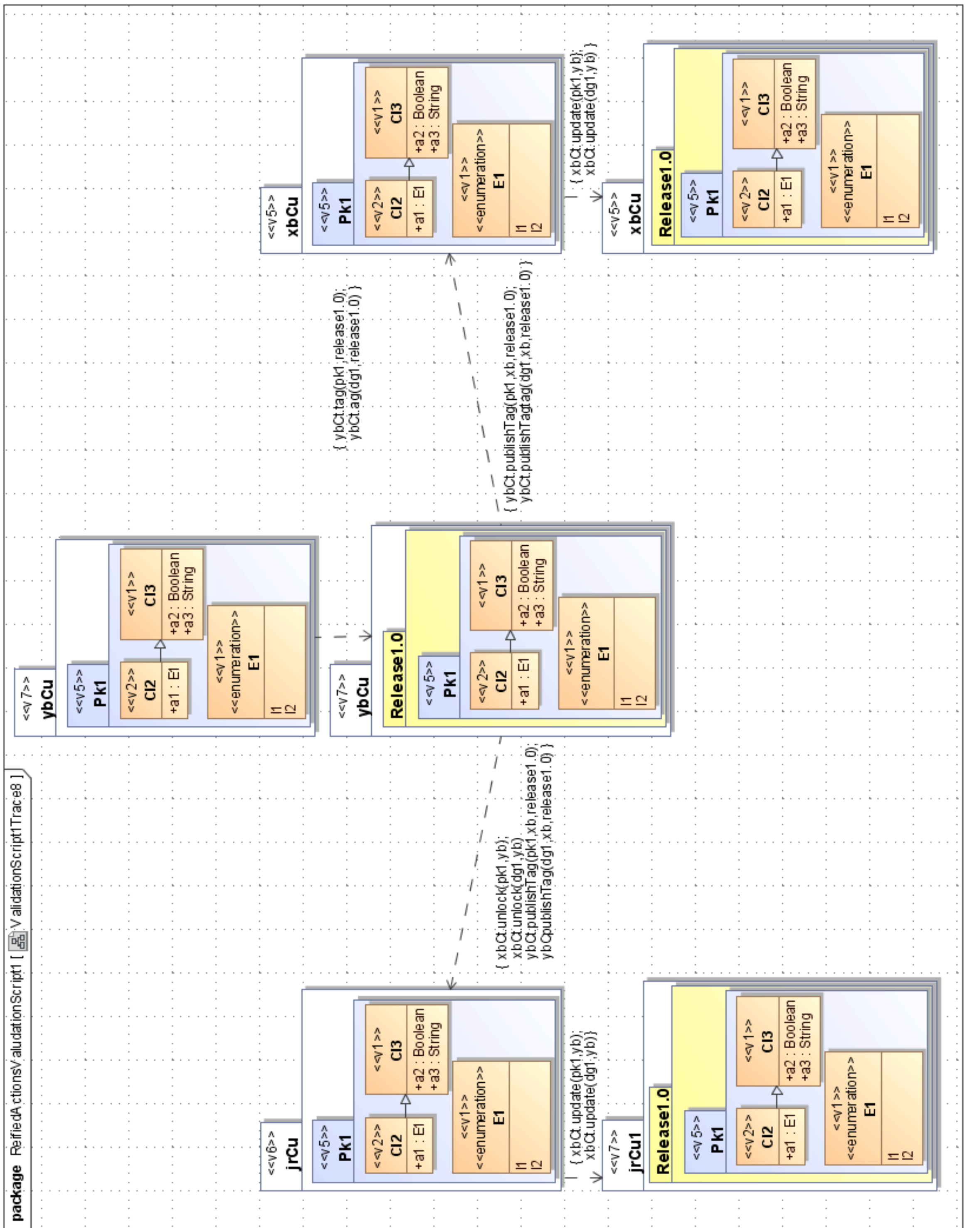

package ReifiedActionsValudationScript1 [ValidationScript1Trace8]
<<v7>> ybCu
<<v5>> Pk1
<<v2>> Cl2
+a1 : E1
<<v1>> Cl3
+a2 : Boolean
+a3 : String
<<v1>> <<enumeration>> E1
l1
l2
<<v7>> ybCu
Release1.0
{ ybCt.tag(pk1,release1.0); ybCt.ag(dg1,release1.0) }
{ ybCt.publishTag(pk1,xb,release1.0); ybCt.publishTagtag(dg1,xb,release1.0) }
{ xbCt.unlock(pk1,yb); xbCt.unlock(dg1,yb); ybCt.publishTag(pk1,xb,release1.0); ybCtpublishTag(dg1,xb,release1.0) }
<<v5>> xbCu
{ xbCt.update(pk1,yb); xbCt.update(dg1,yb) }
<<v6>> jrCu
<<v7>> jrCu1

Figure 17: Blessed collaborative unit owner creates first release tag and publishes it.

7.9 CONCLUSION ON THE GALAXY FRAMEWORK USAGE SCRIPT

The script just presented illustrated, on a simple example, the usage of the proposed galaxy framework interfaces to MDE case tools. Although concise, this use case example is a thorough validation script for these interfaces since it contains instances of calls to 24 of the 31 operations provided by the read-write interfaces. The only ones not covered are commitg, the centralized alternative to commitl, removals duals of addition operations in the GalaxyAdmin API (rmUser, rmProj, rmProjPartic, rmPullRemote, rmPushRemote) which side-effect are trivial and the rebase operation of the GalaxyLocalRevisionHistory API which is only used to simplify histories of old and very branchy projects and thus could not be illustrated on a simple example. At each step of the script, we showed the MDE CASE tool provided view of the project revision for each of its participant. To build these views, the MDE CASE tool calls the read-only operation of the GalaxyQuery API.

The operations of the GalaxyLocalRevision, GalaxyRevisionNotification and GalaxyModelRevision APIs validated by this script are all realized by calls to corresponding operations of: the generic association class CollabUnit mediated by calls to concrete specializations of the abstract class RevisionStrategy. Therefore, this script also validates our definition of the galaxy collaborative unit which is the focus of this deliverable.

In the next and last section of it, we give a concrete example of a model fragmentation strategy and we show instances of the artifact data structures that persistently store model element and views for revision control purposes in a collaborative unit when following this strategy.

Figure 18: Blessed collaborative unit owner creates first release tag and publishes it.

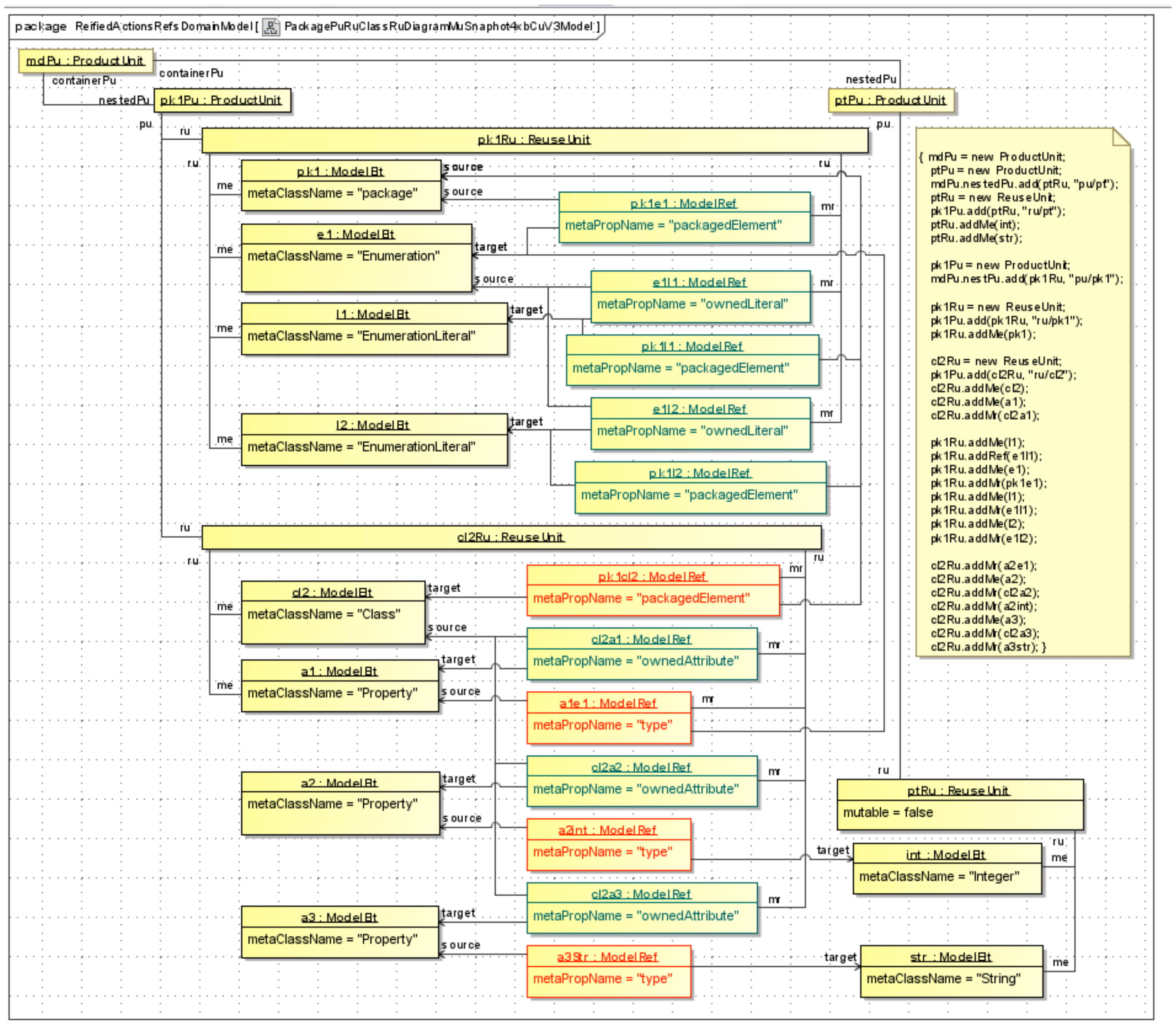
8. AN EXAMPLE OF MODEL FRAGMENTATION STRATEGY

In this section, we provide and explain examples of the internal representation stored in a collaborative unit of a model and one view of it. More precisely, we show the artifact objects stored in revisions v3 and v4 of xb's collaborative unit in the example script of the previous sections. The diagram and model elements contained in these versions are shown on the left-side of Figure 12, in the middle and at the bottom. This figure provides the *external, modeler's perspective* that xb gets of his collaborative unit content through his CASE tool connected to the galaxy instance. In contrast, Figure 19, Figure 20 and

Figure 21 below provides the *internal, system perspective* of the collaborative unit data structure that persistently store this content for revision control purposes. More precisely, Figure 19 shows the artifacts storing the model elements shown in package xbCu stereotyped with <<v3>> in Figure 12, Figure 20 shows the artifact storing how the diagram shown in xbCu stereotyped with <<v3>> displays these model elements and

Figure 21 shows the artifacts storing the the model elements shown in package xbCu stereotyped with <<v4>> in Figure 12. Since the changes made from revision v3 to revision v4 of xb's collaborative unit only alter the model elements that it stores and not the way these elements are displayed in the diagram, we omitted, for conciseness' sake, the diagram artifacts for revision v4.

As explained in section 6.7, a revision strategy is decomposed into a model fragmentation strategy, a model merging strategy and an inconsistency audit strategy. Since the latter two define the behavior of a specific instantiation of the galaxy framework, in concert with the generic behavior specified in the operations of the generic association class CollaborativeUnit, they remain beyond the scope of this deliverable D2.1. These strategies will be the focus of the next deliverable D2.2. In contrast, a model fragmentation strategy defines how to project model elements and views into the three generic artifact classes that we proposed in the definition of the galaxy collaborative unit: product, reuse and diagram units. Such strategy defines the structural modeling approach chosen when instantiating the abstract galaxy framework into a concrete galaxy RCS. Illustrating the definition and usage of one such model fragmentation strategy thus belongs to the scope of deliverable D2.1.


package ReifiedActionsRefsDomainModel [PackagePuRuClassRuDiagramMuSnaphot4xbCuV3Model]
mdPu : ProductUnit
containerPu
nestedPu
pk1Pu : ProductUnit
ptPu : ProductUnit
pu
ru
pk1Ru : ReuseUnit
pk1 : ModelElt
metaClassName = "package"
source
pk1e1 : ModelRef
metaPropName = "packagedElement"
mr
me
e1 : ModelElt
metaClassName = "Enumeration"
target
e1l1 : ModelRef
metaPropName = "ownedLiteral"
l1 : ModelElt
metaClassName = "EnumerationLiteral"
pk1l1 : ModelRef
metaPropName = "packagedElement"
e1l2 : ModelRef
metaPropName = "ownedLiteral"
l2 : ModelElt
metaClassName = "EnumerationLiteral"
pk1l2 : ModelRef
metaPropName = "packagedElement"
cl2Ru : ReuseUnit
pk1cl2 : ModelRef
metaPropName = "packagedElement"
cl2 : ModelElt
metaClassName = "Class"
cl2a1 : ModelRef
metaPropName = "ownedAttribute"
a1 : ModelElt
metaClassName = "Property"
a1e1 : ModelRef
metaPropName = "type"
cl2a2 : ModelRef
metaPropName = "ownedAttribute"
a2 : ModelElt
metaClassName = "Property"
a2int : ModelRef
metaPropName = "type"
cl2a3 : ModelRef
metaPropName = "ownedAttribute"
a3 : ModelElt
metaClassName = "Property"
a3str : ModelRef
metaPropName = "type"
ptRu : ReuseUnit
mutable = false
int : ModelElt
metaClassName = "Integer"
str : ModelElt
metaClassName = "String"

```
{ mdPu = new ProductUnit;
  ptPu = new ProductUnit;
  mdPu.nestedPu.add(ptRu, "pu/pt");
  ptRu = new ReuseUnit;
  pk1Pu.add(ptRu, "ru/pt");
  ptRu.addMe(int);
  ptRu.addMe(str);

  pk1Pu = new ProductUnit;
  mdPu.nestPu.add(pk1Ru, "pu/pk1");

  pk1Ru = new ReuseUnit;
  pk1Pu.add(pk1Ru, "ru/pk1");
  pk1Ru.addMe(pk1);

  cl2Ru = new ReuseUnit;
  pk1Pu.add(cl2Ru, "ru/cl2");
  cl2Ru.addMe(cl2);
  cl2Ru.addMe(a1);
  cl2Ru.addMr(cl2a1);

  pk1Ru.addMe(l1);
  pk1Ru.addRef(e1l1);
  pk1Ru.addMe(e1);
  pk1Ru.addMr(pk1e1);
  pk1Ru.addMe(l1);
  pk1Ru.addMr(e1l1);
  pk1Ru.addMe(l2);
  pk1Ru.addMr(e1l2);

  cl2Ru.addMr(a2e1);
  cl2Ru.addMe(a2);
  cl2Ru.addMr(cl2a2);
  cl2Ru.addMr(a2int);
  cl2Ru.addMe(a3);
  cl2Ru.addMr(cl2a3);
  cl2Ru.addMr(a3str); }
```

Figure 19: Artifacts representing *revision* 3 of the validation script's *model* in xb's collaborative unit (cf. Figure 12)".

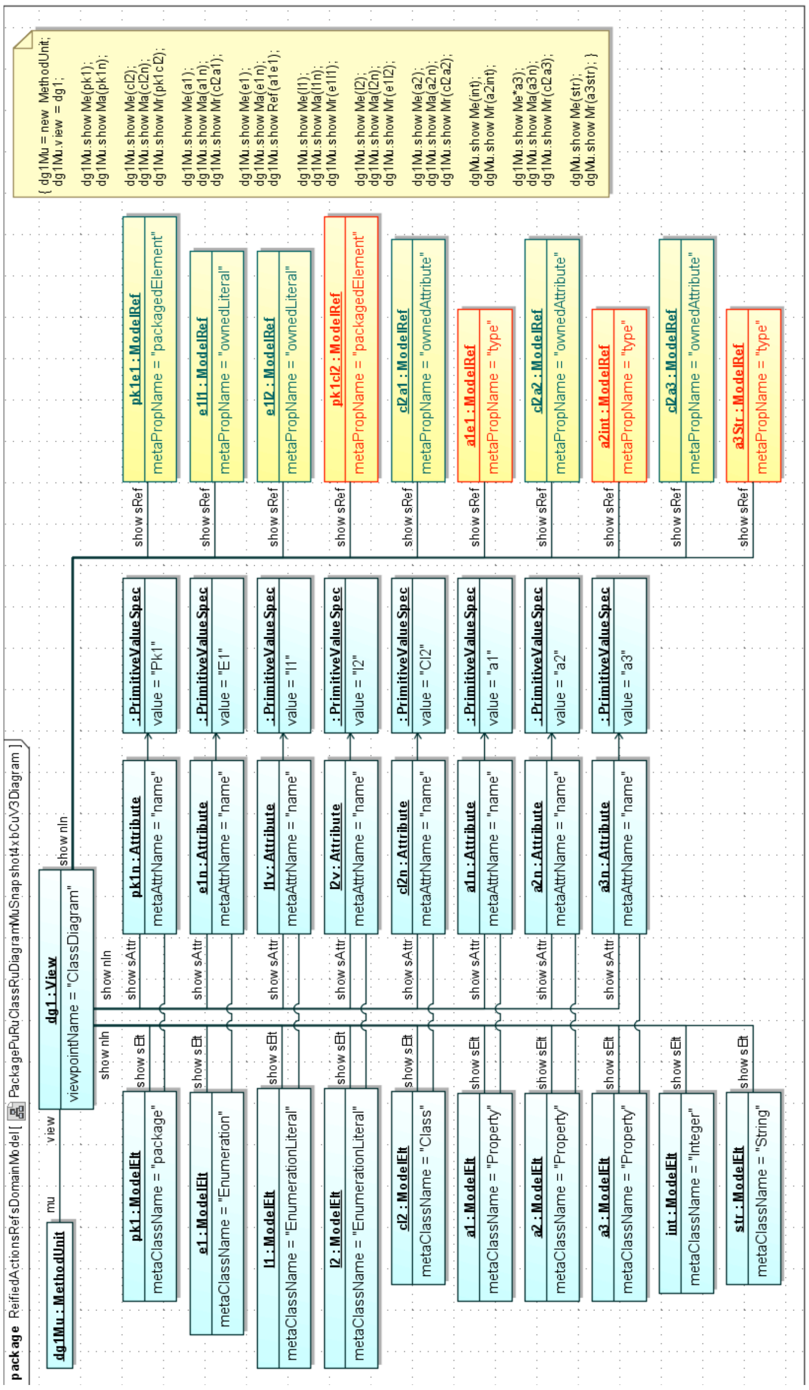
The strategy we choose here to present can be intuitively specified in natural language as follows:

1. Create a root product unit for the whole model;
2. Create one product unit and one reuse unit inside this product unit to store the UML2 primitive types;
3. Create a separate product unit for each different UML2 component or package in the model;
4. Reflect nesting relationships of UML2 components or packages inside larger gain components or packages by nesting relationships between the product units created for these components or packages;
5. Create a separate method unit for each diagram in the model
6. Create a separate reuse unit for each different non-relationship classifier (*e.g.,* classes, interfaces, components) and package in the model;
7. Nest the reuse unit storing a component or package element inside the product unit created for this component or package element;
8. Nest the reuse units storing all the model elements packaged into a UML2 component or package inside the product unit created for this component or package;
9. Store the features of a classifier inside its reuse unit;
10. Store the generalization between two classifiers inside the reuse unit of the more specific classifier;
11. Store the association from a source classifier to a target classifier inside the reuse unit of the source classifier;
12. Store the references between two model elements stored in the same reuse unit in this reuse unit;
13. Store the reference of meta-association *specific* and *type* from a model element *me1* of reuse unit *ru1* to a model element *me2* of reuse unit *ru2* ≠ *ru1* in *ru1*;
14. Store the reference of meta-association *packagedElemen*t from a model element *me1* of reuse unit *ru1* to a model element *me2* of reuse unit *ru2* ≠ *ru1* in *ru2*.

The artifact resulting from applying this model fragmentation strategy on the revision v3 of the model stored in xb's collaborative unit (shown in Figure 12) is shown in Figure 19 It is an object diagram containing instances of the classes ProductUnit and ReuseUnit defined in Figure 4. Recall from section 4 that we propose to structure galaxy artifacts are organized to form a containment tree. Following the fragmentation strategy just described, the root object of this containment tree is a product unit, called mdpu, which contains all the other artifacts of the project's model. It contains two nested product units, ptpu, which contains the model-independent UML2 primitive type and pk1pu which contain the package pk1. Ptpu contains only one reuse unit ptru which in turns contains the UML2 primitive type model elements. In the diagram of Figure 19 we only show the two primitive types appearing in the revision v3 of xb's collaborative unit, i.e., integer int and string str. Pk1pu contains one reuse unit pk1ru which contains the model element pk1 and one reuse unit cl2ru which contains the class cl2. In addition to pk1, pk1ru also contains three other model elements, enumeration e1 and its two enumeration literals l1 and l2. It also contains five references pk1e1 from pk1 to e1, pk1l1 from pk1 to l1, pk1l2 from pk1 to l2, e1l1 from e1 to l1 and e1l2 from e1 to l2. In addition to cl2, cl2ru also contains three other model elements, a1, a2 and a3 one per attribute of cl2. It also contains seven references: pk1cl2 from pk1 to cl2, cl2a1 from cl2 to a2, cl2a2 from cl2 to a2, cl2a3 from cl2 to a3, a1e1 from a1 to e1, a2int from a2 to int and a3str from a3 to str.

On the top right of Figure 19, we show the product unit and reuse unit action sequences that represents the construction trace of the units displayed in the rest of the figure. Task T2.2 of the project, we will investigate what representation is more scalable to be passed for change notification purposes among collaborative units: the structural snapshot, a structural diff from the previous revision snapshot, the behavioral diff consisting of the artifact action sequence or some combination of the three. One last thing to explain about Figure 19, is that references internal to a reuse unit are highlighted by a green frame and font, whereas references across reuse units are highlighted by a red frame and font. In the artifact base shown in this figure, there are thus eight internal references and four cross-references. Minimizing cross-reference is a simple general scalability heuristics when devising a model fragmentation strategy, since changing an internal reference can be done by loading into main memory and searching inside a single reuse unit, whereas a changing a cross-reference involves loading and searching two such units.

Having shown an illustrative example of persistent storage format for model elements in a collaborative unit, we now discuss an illustrative example of persistent storage format to keep the information of which model elements, references and attributes are displayed in each view of the model. The example is given in Figure 20. It is an object diagram that contains an instance of the class MethodUnit defined in Figure 4. This instance stores the class diagram that shows all model elements in revision v3. The left side of Figure 20 contains a sequence of method unit actions which constitute the construction trace of the method unit shown in the same figure. It is the translation, at the internal collaborative unit layer, for the model fragmentation strategy specified at the beginning of this section, of the view actions executed by the CASE tools of yb, jr and xb at the galaxy framework API layer, and which collectively resulted into the class diagram shown in revision v3 of xb's collaborative unit. Similarly than for the model artifacts, such method unit action sequence constitutes an alternative representation of the method unit object network shown in the same figure. Either one or some clever combination of the two could be passed among collaborative units for scalable notification of changes in the views and diagrams that correspond not to changes in the model but only to visualization choice changes over that model. The scalability strengths and drawback of each representation will be investigated in T2.2



```
{ dg1Mu = new MethodUnit;
  dg1Mu.view = dg1;

  dg1Mu.show Me(pk1);
  dg1Mu.show Ma(pk1n);

  dg1Mu.show Me(cl2);
  dg1Mu.show Ma(cl2n);
  dg1Mu.show Mr(pk1cl2);

  dg1Mu.show Me(a1);
  dg1Mu.show Ma(a1n);
  dg1Mu.show Mr(cl2a1);

  dg1Mu.show Me(e1);
  dg1Mu.show Ma(e1n);
  dg1Mu.show Ref(a1e1);

  dg1Mu.show Me(l1);
  dg1Mu.show Ma(l1n);
  dg1Mu.show Mr(e1l1);

  dg1Mu.show Me(l2);
  dg1Mu.show Ma(l2n);
  dg1Mu.show Mr(e1l2);

  dg1Mu.show Me(a2);
  dg1Mu.show Ma(a2n);
  dg1Mu.show Mr(cl2a2);

  dgMu.show Me(int);
  dgMu.show Mr(a2int);

  dg1Mu.show Me*a3);
  dg1Mu.show Ma(a3n);
  dg1Mu.show Mr(cl2a3);

  dgMu.show Me(str);
  dgMu.show Mr(a3str); }
```

Figure 20: Artifacts representing *revision 3* of the validation script *diagram* in xb's collaborative unit (cf. Figure 12)".

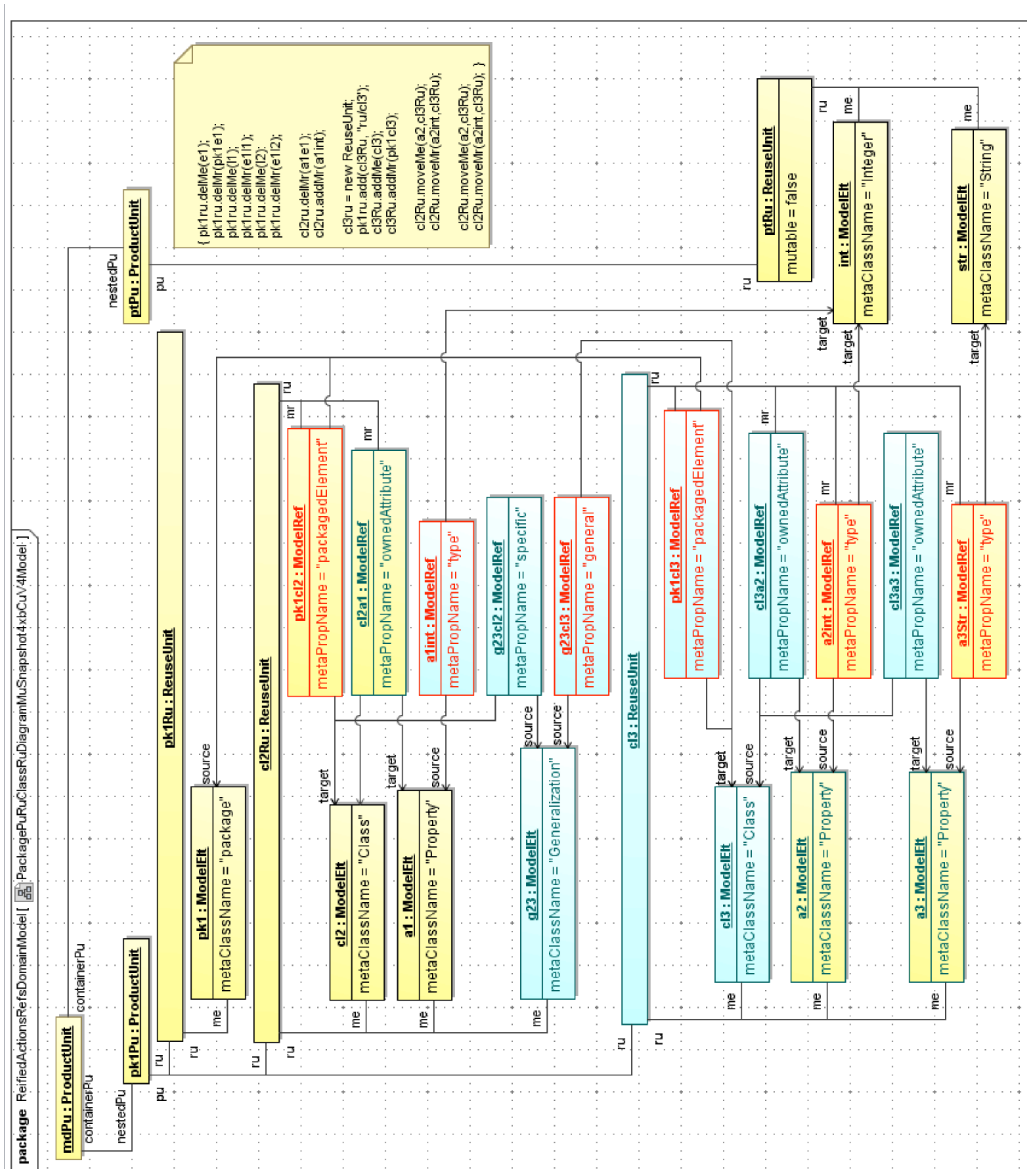
To conclude our presentation of illustrative examples of artifact data structures internal to collaborative units, we show in Figure 21 the artifacts in xb's collaborative for its next revision v4 originally shown in Figure 12. These updated artifact object network results from the execution on the artifact shown in Figure 19 (corresponding to revision v3) of the artifact action sequence shown on the left side of Figure 21. This action sequence is itself the translation of the model action sequence from revision v3 to revision v4 of xb's collaborative unit show in in Figure 12, given the model fragmentation strategy specified at the beginning of this section. Let us compare, on the one hand, the model element, galaxy API level differences between revisions v3 and v4, and on the other hand, the artifact, collaborative unit level differences between these two revisions.

At the model element level the differences are:

1. attribute *a1* type changed from enumeration type *E1* to primitive type integer;
2. no longer needed enumeration *E1* (and its two literals) deleted;
3. new class *cl3* created;
4. attributes *a2* and *a3* moved from class *cl2* to class *cl3*;

At the artifact level these differences translate into:

1. reference *a1e1* deleted from reuse unit *cl2ru* and new reference *a1int* created in *cl2ru*;
2. model elements *e1, l1, l2* and the references from and to them (*i.e., pk1a1, e1l1, pk1l1, e1l2, pk1l2*) deleted from reuse unit *pk1ru*;
3. reuse unit *cl3ru* created and added to product unit *pk1ru*; model element *cl3* and reference *pk1cl3* from *pk1* to *cl2* added to *cl3ru*;
4. model elements *a1* and *a2* and references to and from them moved from reuse unit *cl2ru* to reuse unit *cl3ru*.



```
{ pk1ru.delMe(e1);
pk1ru.delMr(pk1e1);
pk1ru.delMe(l1);
pk1ru.delMr(e1l1);
pk1ru.delMe(l2);
pk1ru.delMr(e1l2);

cl2ru.delMr(a1e1);
cl2ru.addMr(a1int);

cl3ru = new ReuseUnit;
pk1ru.add(cl3Ru, "ru/cl3");
cl3Ru.addMe(cl3);
cl3Ru.addMr(pk1cl3);

cl2Ru.moveMe(a2,cl3Ru);
cl2Ru.moveMr(a2int,cl3Ru);

cl2Ru.moveMe(a2,cl3Ru);
cl2Ru.moveMr(a2int,cl3Ru); }
```

Figure 21: Artifacts representing *revision 4* of the validation script's *model* in xb's collaborative unit (cf. Figure 12)".

9. WHEN IS A STRATEGY SCALABLE?

A strategy has to be scalable in order to manage the revision control of huge models. From a theoretical point of view, a strategy is scalable if the size of the artifact remains constant (in order of magnitude) with the model size and if the number of messages sent to deal with revision control is bounded. From a pragmatically point of view, a strategy is scalable if the artifacts and the messages can be handled by the underlying Revision Control System (RCS) using an acceptable amount of resources and in an acceptable time frame.

9.1 DEFINITION

To measure the scalability of a strategy, we define the following concepts:

- *Model* corresponds to the whole model that is collaboratively edited in a galaxy;
- *Size(Model)* corresponds to the size of a model, in number of model elements;.
- *Artifact* corresponds to the set of artifacts used by a galaxy.
- *Product* corresponds to the product units used by a galaxy
- *Reuse* corresponds to the reuse units used by a galaxy
- *Method* correspond to the method units used by a galaxy
- *Size(Artifact(i))* corresponds to the size of the i^{th} artifact stored in a galaxy's collaborative units. It is equal to:
  - 0 if it the artifact is a product unit.
  - the number of model elements that it contains, if the artifact is a reuse unit.
  - the number of model elements that it references, if the artifact is a method unit
- *VuActions* corresponds to the sequence of model (elements) and view edition actions executed by a developer;throuh his (her) CASE tool connected to a galaxy framework instance;
- *AfActions* corresponds to a sequence of artifacts actions sequence executed by a galaxy framework instance in reaction to *VuActions;*
- *Size(VuActions)* is the length of *VuActions;*
- *Size(AfActions)* is the length of *AfActions;*

- *Artifact(AfActions)* is to the set of artifacts appearing as arguments of an action in *AfActions;*
- *MAS (Maximum Artifact Size)*.is the maximum number of model element in the largest galaxy artifact;

MEM (Maximum number of Exchanged Messages) is the maximum number of messages exchanged among collaborative units of a galaxy to reflect an update to a new version of a model fragment or view?;

- *card(Artifact(s))* is the number of artifacts needed to store a given model and its various view following revision strategy s

9.2 THEORETICAL SCALABILITY

In order to be theoretically scalable, the size of artifacts should be far inferior to the size of the whole model. Otherwise, a few artifacts will contain the majority of the model elements, which presents the following scalablity problems:

- Loading in main memory any model element contained in those overly large artifacts becomes too space and time consuming;
- Locking any model element contained in those overly large artifacts will result in locking a a large fragment of the global model , namely all the model elements the ones that are contained in the same artifacts than the one being edited by one developer; this results in locking most developers out of their current tasks most of the time;
- Exchanging any model element contained in those overly large artifacts will then result in prohibitively costly exchange of needlessly large sets of model elements;
- Versioning any model element contained in those overly large artifacts will result in needlessly versioning all model elements contained in the same artifact.

Consequently, if the size of the artifacts is not bound, the approach does not scale up.

Quantitatively, this requirement is expressed by the first Theoretical Scalability Property (TSP1), which defines a bound:

(TSP1) $|\mathrm{Max}(\mathit{Size}(\mathit{Artifact}(i)) - \mathrm{MAS}| \le \varepsilon$

This is a necessary but not sufficient theoretical scalability condition. In addition, another parameter should be bound: the number of artifact actions performed by a galaxy framework configured with a given strategy in response to model modifications *typically* carried out by developers. If this number is not bound then each time a developer commit its last modifications, two many messages will be exchanged among the collaborative units of the framework to update the model within an acceptable response time, Here we talk about typical modifications, i.e., those that are actually commonly performed by developers in real, industrial MDE projects. We do not worry here about any theoretically possible sequences given the action vocabulary we provided in section 5.

Quantitatively, this second scalability condition is expressed by the second theoretical scalability property (TSP2):

(TSP2) $\forall$ *VuActions* $\in$ TypicalActions $|\mathrm{Size}(\mathit{AfActions}) - \mathrm{MEM}| \le \varepsilon$

9.3 PRAGMATIC SCALABILITY

From a pragmatic point of view, existing revision control systems fail in managing a huge set of artifacts. As a consequence, a strategy is more pragmatically scalable than another one if it needs fewer artifacts. .

Quantitatively, this is expressed by the first pragmatic sclalability property (PSP1) :.

Let s1, s2 be two strategies, s1 is more pragmatically scalable than s2 if and only if

Card(*Artifact(s1))* $_c$< Card(*Artifact(s2))*

Additionally, existing revision control systems fail in scalably versioning tightly coupled artifacts. In such cases, one change made to a model by a developer may require to be propagated into changes to many artifacts, the revision control system must then open most artifacts in order to perform conflict detection during a merge.

Quantitatively, this is expressed by the second Pragmatic Scalability Property (PSP2):

Let s1, s2 be two strategies, s1 is more pragmatically scalable than s2 if and only if

$\mathrm{Card}(Artifact(AfActions))_{s1} < \mathrm{Card}(Artifact(AfActions))_{s2}$

So while PSP1 compares the structural scalability of two strategies, PSP2 compares their behavioral scalability. PSP1 measures the impact of a change in terms of the number of data structures onto which the consequences of the change need to be propagated. In contrast, PSP2 measures that impact in terms of the number of operations to execute on these data structures to perform such propagation.

9.4 EXAMPLE

This section assesses the scalability of the revision strategy given as an example provided in sections 7 and 8.

9.4.1 Theoretical scalability

This strategy is theoretically scalable since it verifies both TSP1 and TSP2.

Regarding (TSP1):

- $\mathrm{Max}(Size(Artifact_i)) = F$
  - Where F is the maximum number of feature per classifier.

We can reasonably state that MAS is a constant (a class contains at most 100 model elements, then $| F - 100 | \leq \varepsilon$).

Regarding (TSP2):

- Size(*AfActions*) = K * *VuActions*
    - Where K is a constant. Indeed, for each action in VuAction, at most K actions are needed.

We can reasonably state that MEM is a constant as a typical change is a bounded sequence of actions (K_VU), then ∣ Size(*AfActions*) – K*K_VU ∣ ≤ ε)..

9.4.1 Pragmatically scalability

As the pragmatically scalability, we can state that:

- Card(Artifact) = 1(PU Project) + P*(PU Package) + 1 (PU BuiltIn) + 1 (RU BuiltIn) + C*(RU Non Relationship Classifier) + D * (MU Diagram) = 3 + P + C + D
    - Where P is the number of packages, C is the number of non relationship classifier and D is the number of diagram.
- Card*(Artifact(AfActions))* will depend on how much classifiers are classically modified conjointly by developers.

10. REFERENCES

[1] Atkinson, C., Bayer, J., Bunse, C., Kamsties, R., Laitenberger, O., Laqua, R., Muthig, D., Paech. Barbara, Wust, J. and Zettel, J. *Component-based product line engineering in UML*. Addison-Weslei. 2001.

[2] Akinson, C., Stoll, D. *Orthographic modelling environment*. In proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08). 2008. Budpaest, Hungary.

[3] Atkinson, C., Gutheil, M and Kennel, B. *A flexible infrastructure for multi-level language engineering*. IEEE Transactions on Software Engineering, 35, 2009.

[4] 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.

[5] Canonical Ltd. The Bazaar wiki. http://wiki.bazaar.canonical.com/

[6] Canonical Ltd. Bazaar 2.x benchmarking results. http://wiki.bazaar.canonical.com/Benchmarks

[7] Canonical Ltd. Bazaar supported workflows. http://wiki.bazaar.canonical.com/Workflows

[8] Chacon. S. *Pro Git*. APress. 2009.

[9] Chacon, S. Why is Git better than X. http://whygitisbetterthanx.com/

[10] Collins-Sussman B., Firtzpatrick, B. and Pilato, M.C. *Version Control with Subversion*. O'Reilly. 2008.

[11] Kruchten, P.B. *The Rational Unified Process: an introduction (3rd Ed).* Addison-Wesley. 2003.

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

[13] Object Management Group. The XML Metadata Interchange.

www.omg.org/technology/documents/formal/xmi.htm.

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

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

[16] Murta, L., Dantas, H., Oliveira, H., Lopes, L. and Werner, C. *Odyssey-SCM: An integrated software configuration management infrastructure for UML models.* Science of Computer Programming. 65(3). 2007. Elsevier.

[17] Murta, L., Corrêa, C., Prudêncio, J.G. and Werner, C. Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System. In proceedings of the ACM/IEEE ICSE Workshop on Comparison and Versioning of Software Models (CVSM08), Leipzig, Germany, May 2008, pp. 25-30.

[18] 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.

[19] Steinberg, D., Budinsky, F., Paternostro, M. and Merks, Ed. *Eclipse Modeling Framework* (2nd Ed.). Addison-Wesley. 2008.