©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

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

Deliverable D2.1: Collaborative Unit Definition

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

	NAME	PARTNER	DATE
WRITTEN BY	J. Robin	LIP6	
	X. Blanc	LIP6	
REVIEWED BY	Y. Bernard	Airbus	
	F. Racaru	Akka	
	B. Coulette	IRIT	
	P. Vlaeminck,	Softeam	
	F. Reynaud	Softeam	

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

RECORD OF REVISIONS

ISSUE	DATE	EFFECT ON		REASONS FOR REVISION
		PAGE	PARA	
01A				Création du document

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

TABLE OF CONTENTS

1. INTRODUCTION 13

2. APPROACH: MDE AND SEPARATION OF CONCERNS 16

3. DEFINITION OF THE COLLABORATIVE UNIT 17

3.1 CLASS NAMED ENTITY 20

3.2 CLASS GALAXY FRAMEWORK 20

3.2.1 Definition 20

3.2.2 Properties 23

3.3 CLASS PROJECT 23

3.3.1 Definition 23

3.3.2 Properties: 23

3.4 CLASS PARTICIPANT 23

3.4.1 Definition 23

3.4.2 Properties: 24

3.5 ASSOCIATION CLASS COLLAB UNIT 24

3.5.1 Definition 24

3.5.2 Properties: 24

3.5.3 Components 25

3.5.4 Operations 25

3.6 ABSTRACT CLASS HISTORY UNIT 30

3.6.1 Definition 30

3.6.2 Properties 30

3.7 ABSTRACT CLASS REVISION UNIT 31

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

3.7.1 Definition 31
3.7.2 Properties 31
3.8 ABSTRACT CLASS ARTIFACT 32
3.8.1 Definition 32
3.8.2 Properties 32
3.8.3 Operations 32
3.9 ABSTRACT CLASS ATOMIC ARTIFACT 33
3.10 ABSTRACT CLASS COMPOSITE ARTIFACT 33
3.10.1 Definition 33
3.10.2 Properties 33
3.10.3 Operations 33
3.11 CLASS COMMIT 34
3.11.1 Definition 34
3.11.2 Properties 34
3.11.3 Operations 35
3.12 ABSTRACT CLASS AFDIFF 35
3.12.1 Definition 35
3.12.2 Properties 35
3.12.3 Operations 36
3.13 ABSTRACT CLASS AFACTION 36
3.13.1 Definition 36
3.13.2 Operations 36
3.14 CLASS TAG 36
3.14.1 Definition 36

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

3.14.2 Properties 36

3.14.3 Operations 37

3.15 CLASS BRANCH TAG 37

3.15.1 Definition 37

3.15.2 Properties 37

3.15.3 Operations 37

3.16 CLASS LOCK TAG 37

3.16.1 Definition 37

3.16.2 Properties 37

3.16.3 Operations 37

4. DEFINITION OF VIEWS AND MODEL FRAGMENTATION MECHANISMS 38

4.1 HOW TO STRUCTURE MODEL ELEMENTS AND VIEWS ON THEM FOR REVISION CONTROL PURPOSES? 38

4.2 CLASS MODEL ELT 41

4.2.1 Definition 41

4.2.2 Properties 41

4.2.3 Operations 42

4.3 CLASS ATTRIBUTE 42

4.3.1 Definition 42

4.3.2 Properties 42

4.3.3 Operations 42

4.4 CLASS MODEL REF 43

4.4.1 Definition 43

4.4.2 Properties 43

4.4.3 Operations 43

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

4.5 CLASS VIEW **44**

4.5.1 Definition 44

4.5.2 Properties 44

4.5.3 Operations 44

4.6 CLASS METHOD UNIT **45**

4.6.1 Definition 45

4.6.2 Properties 45

4.6.3 Operations 45

4.7 CLASS REUSE UNIT **45**

4.7.1 Definition 45

4.7.2 Properties 45

4.7.3 Operations 46

4.8 CLASS PRODUCT UNIT **46**

4.8.1 Definition 46

4.8.2 Components 46

4.8.3 Properties 47

4.8.4 Operations 47

5. ACTIONS TO EXECUTE DURING THE COLLABORATIVE UNIT LIFE CYCLE **47**

6. CONTROLLING THE COLLABORATIVE UNIT LIFECYCLE FROM AN MDE CASE TOOL **52**

6.1 APPROACH: DECOUPLING AND SEPARATION OF CONCERNS **53**

6.2 THE GALAXY QUERY API **53**

6.3 THE GALAXY ADMIN API **54**

6.4 THE GALAXY LOCAL REVISION HISTORY API **55**

6.5 THE GALAXY REVISION NOTIFICATION **55**

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

6.6 THE GALAXY MODEL REVISION API 55

6.7 THE GALAXY CLASSES REALIZING THE GALAXY APIS 56

7. AN EXAMPLE OF COLLABORATIVE UNIT LIFECYCLE 58

7.1 CONFIGURING THE COLLABORATIVE WORKFLOW AMONG GALAXY USERS 59

7.2 STEP 1: BLESSED REPOSITORY GATEKEEPER CREATES FIRST VERSION OF THE MODEL, PUBLISHES IT AND THE COLLABORATORS CLONE IT 62

7.3 STEP 2: CONFLICT FREE CONCURRENT REVISIONS OF THE MODEL BY COLLABORATORS 64

7.4 STEPS 3 CONFLICTING CONCURRENT REVISIONS OF THE MODEL BY COLLABORATORS 68

7.5 STEP 4 BLESSED COLLABORATIVE UNIT GATE KEEPER DELEGATES ERROR RESOLUTION BY CREATING BRANCHES AND PUBLISHING THEM 80

7.6 STEP 6 ONE COLLABORATOR MERGES DESIGN CHOICES FROM BOTH BRANCHES INTO ONE, DELETES THE OTHER AND PUBLISHES IT 85

7.7 STEP 7: COLLABORATOR AGREES WITH CHANGES MADE BY THE OTHER 87

7.8 STEP 8 BLESSED COLLABORATIVE UNIT OWNER CREATES RELEASE1.0 I 87

7.9 CONCLUSION ON THE GALAXY FRAMEWORK USAGE SCRIPT 91

8. AN EXAMPLE OF MODEL FRAGMENTATION STRATEGY 91

9. WHEN IS A STRATEGY SCALABLE? 100

9.1 DEFINITION 100

9.2 THEORETICAL SCALABILITY 101

9.3 PRAGMATIC SCALABILITY 102

9.4 EXAMPLE 103

9.4.1 Theoretical scalability 103

9.4.1 Pragmatically scalability 104

10. REFERENCES 104

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

TABLE OF APPLICABLE DOCUMENTS

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

TABLE OF REFERENCED DOCUMENTS

N°	TITLE	REFERENCE	ISSUE
R1	Galaxy glossary		
R2			
R3			
R4			

ACRONYMS AND DEFINITIONS

Except if explicitly stated otherwise the definition of all terms and acronyms provided in [R1] is applicable in this document. If any, additional and/or specific definitions applicable only in this document are listed in the two tables below.

Acronymes

ACRONYM	DESCRIPTION

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

Definitions

TERMS	*DESCRIPTION*

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

1. INTRODUCTION

This document presents D2.1., *i.e.* the first deliverable of the second work package of the project. The goal of this deliverable is to provide a general, high-level specification of the concept of collaborative unit in large scale model-driven software engineer projects.

The in scope of this specification was defined in the deliverable D0.1.1. (Project Presentation) which states three main tasks for D2.1.

1. *Definition of the model fragmentation mechanism*: The management of huge models needs a model fragmentation mechanism; this mechanism must allow project participants to work only on small model fragments relevant to their current task, while insuring the consistency of the overall model from which these fragments are extracted and then merged; section **4** defines such a mechanism; section 8 presents a concrete example of its application;
2. *Definition of user workspace*: in the context of model driven collaborative development, each developer works on its own workspace composed only of model elements that are relevant for him; section 4 defines the concept of workspace and its relationship with model and model fragment;
3. *Definition of collaborative unit life cycle*: a collaborative unit is the data structure that underlies the storage of model fragments and views; section 3 defines the collaborative unit and its operations; section 5 and 6 define its life cycle, *i.e.,* how calls to its operations affect its states and vice-versa; section 7 gives illustrative examples of collaborative units and collaborative uni lifecycles.

What is meant in practice by the adjective “huge” in the above description was quantitatively estimated in deliverable D1.1. It mentions models developed by up to 150 collaborators and comprising up to 1400 distinct entities and 5500 relationships among these entities. D1.1. also suggests five metrics for collaborative MDE: (a) the persistent storage space requirement for the model revision history, (b) the message size for the model revision synchronization operations, (c) the computer processing time for the automated model revision control operations such as *commit,*

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

update or *branch*, (d) the human processing time for manual model revision control operations such as semantic conflict resolution and (e) the number of project participants. The relation between the model size metrics and the collaborative unit scalability metrics is not direct. It involves other mediating factors. The first is the data format adopted for model persistent storage and update messages (*e.g.,* XMI [13] files *vs*. Praxis fact files [4] [15]) which defines how many bytes are needed to store each model element and model element reference. The second factor is the metamodel(s) of the modeling language(s) used to represent the model entities and relationships. This metamodel allows estimating the number of model elements and model element references, per domain model entity and model element relationship[1]. The third factor is the number of distinct views [2] that needs to be maintained for a given model in the galaxy, since each view involves storing, exchanging and processing additional metadata that stacks on top of model data. The fourth factor is the number of modification steps registered in the model history executed during the model development history, since it is the whole model revision history that needs to be stored and processed by collaborative units, not merely the current state of the model. The fifth factor is the *usage patterns* of model revision control operations, defined as the relative frequency of calls to *commit, diff, revert, update, branch, merge, tag*, etc.

In addition to potential for *scalability*, another requirements for the collaborative unit specification presented in this document precise enough to constitute a well-founded basis for subsequent project deliverables of WP2 (conceptual model of model-driven collaborative development) and WP4 (distributed work support prototype) that reuse D2.1 as starting point. Versatility is needed

[1] A domain entity such as a UML class consists of many domain elements (*e.g.,* its ownedAttributes, its ownedOperations, its nestedClassifiers, etc.) and many domain references (*e.g.,* the references between the class and its ownedAttributes, ownedOperations, etc). The same is true for domain relationships such a UML association which also consists of many domain elements (*e.g.,* its type, memberEnd, ownedEnd etc.,) and many domain references (e.g., the references between the association and its type, ,its memberEnd, ownedEnd etc.)

[2] View are defined in section 4.5.1.

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

to make sure that the proposed concept of collaborative unit will apply to a wide spectrum of large scale industrial MDE projects. Only then will the Galaxy project makes a significant contribution to MDE in general, and not merely to one of MDE numerous very specific niches.

To achieve such versatility the collaborative unit concept needs to be sufficiently *generic and abstract* so as to as remain independent as possible from specific:

– Modeling languages (*e.g.*, UML, SysML, Petri nets, DLSs (Domain Specific Languages);

– Model-driven software processes (*e.g.*, RUP[11], KobrA [1]);

– Revision control workflow (e.g., centralized with remote commit, centralized with local commit, decentralized with shared mainline, decentralized with human gatekeeper, decentralized with hierarchy of human gatekeepers [7], etc.);

– Revision control repository deployment (*e.g.*, unique central server *vs.* P2P);

– Persistent data structure (*e.g.*, XMI files *vs*. Praxis action fact files *vs*. relational DB table vs. Eclipse Modeling Framework (EMF) [19] objects persistent on Google's cloud);

The out of the collaborative unit specification was defined in the Project Presentation deliverable D0.1.1., which states three main tasks for later tasks in the Galaxy project that take D2.1. as starting point. D2.1 therefore does *not* cover the precise definition of the three main point of T2.2, which are:

- *Collaborative Unit Diff:* the mechanism that will compute differences between collaborative units;
- *Collaborative Unit Merge:* the mechanism that will compute the merging between collaborative units;
- *Balancing the collaboration strategy:* means to relax safety collaboration constraints, and to set the collaboration level regarding safety and flexibility requirements.

D2.1 also does *not* cover the precise definition of:

- model views which is in the scope of T3.1.;
- the specification of the open and flexible architecture, which is within the scope of T4.1;
- the core API of the Galaxy framework, which is also within the positive scope of T4.1;

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- project participant roles within the process used for a development project, which is within the scope of T2.3.
- However, with the intent to serve as a rich starting point for these tasks, D2.1 does define abstract and/or basic versions of these concepts to be elaborated in the appropriate subsequent project tasks

2. APPROACH: MDE AND SEPARATION OF CONCERNS

Following an MDE approach to the elaboration of the present report, we use UML2 diagrams to define and illustrate the concept of Collaborative Unit. Each diagram is accompanied by a text in natural language. This text explains the semantics of the model elements appearing in the diagram together with the motivation for the particular design that it embodies. Each of the next four chapters presents and explains a set of class diagram that focuses on a single concern. Section 3 focuses on the concern of persistent data structures and operations to scalably and collaboratively revises any sort of software artifacts whether following an MDE or code-driven approach. This is where the concept of **collaborative unit** is introduced. Section 4 focuses on the concern of defining model fragments of intermediary grains between, on the one hand, the macro-grain of the whole project megamodel that integrates all the software project models, and, on the other hand, the micro-grain of individual model elements, and diagrams[3]. This section introduces three distinct types of medium grain fragments: **product unit, reuse unit and method unit.** Section 5 focuses on defining a minimal set of **primitive model and view revision actions** in term of which any model or view manipulation can be ultimately decomposed. Section 6 focuses on showing how the galaxy framework data structures defined in the three preceding section allows providing to MDE CASE tools external revision control services through five simple interfaces. This section introduces the important concept of **revision strategy**, which defines how revision actions on model elements and

---

[3] Here and in the rest of the document, we will use the word diagram meaning "the subset of model elements show in the diagram", not meaning the visual concrete syntax of these elements in the diagram.

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

views executed by a developer in a CASE tool, are to be translated into revision actions on the product units, reuse units and method units that group these elements and view for scalable revision control purposes. The definition of a **revision strategy** for a particular class of projects will be the key design task of project infra-structure staff to leverage the galaxy framework to implement the revision control environment of an MDE software project. Section 6 closes the presentation of the galaxy collaborative unit and framework domain model.

The subsequent sections validate this domain model by presenting an illustrative example of its use on a script in which three developers collaborate to construct a simple UML class diagram. Section 7 follows the step-by-step evolution of this diagram in the collaborative units of these three developers from the construction of its initial version to its freezing as a first preliminary release. Each step contains the model edition and revision control action sequence executed by each developer using a CASE tool connected to an instance of the galaxy framework. It thus provides an external, project artifact developer view of such framework instance. In contrast, section 8 defines an internal view of the framework relevant to the project infrastructure support staff that will configure it for a specific project. It defines a simple revision strategy, specific to UML modeling. It then shows two snapshots of the product unit, reuse unit and method unit data structures that represent the model and associated diagrams “under the-hood” of a revision control environment, instance of the Galaxy framework, and which implements the strategy. These two snapshots respectively correspond to the product, reuse and method unit states before and after one of the eight revision steps shown in the preceding section. These snapshots are represented by UML object diagrams.

3. DEFINITION OF THE COLLABORATIVE UNIT

The collaborative unit allows development project participants to collaboratively construct and revise any artifacts to produce at any phase and step of the project. The PIM of the collaborative unit that we propose is given in Figure 1. In essence, it is a workspace in a Revision Control System

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

(RCS) (see section 3.4 of Galaxy deliverable D1.2 for a definition and review of such systems), since it must provide the same services:

1. *History recording*: keeps track of who made which revision to what artifact(s) when and with what purpose;
2. *Revision roll back:* allows seamless backtracking to past revisions of any artifact;
3. *Branch handling*: allows efficient forking and subsequent merging of concurrent development branches
4. *Artifact comparison* (often called diff): point out the differences between two artifacts, generally with respect to a third artifact from which the two being compared independently evolved;
5. *Error detection*: identify the problematic cases of artifact differences.

The collaborative unit concept corresponds to both the local workspaces of an RCS deployed on the machine of a project participant, and the centralized workspace of an RCS that serves as unique reference from which to update the local workspaces and to which commit changes from local workspaces. We define *error* as a concept that subsumes both *conflicts* and *inconsistencies*. A conflict occurs when two model fragments, views or artifacts cannot be automatically merged by the simple union of the nodes and edges from the labeled directed graphs that (respectively) represent them. This is the case for example if the name of the same model element in both fragment is different (*i.e.,* incompatible property value) or if an the origin or destination node of an edge in one fragment has been deleted in another *(i.e.,* a dangling link). A successful automatic merge may result in a well-formed graph which nonetheless contains some inconsistencies.

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

Figure 1: The concept of galaxy collaborative unit

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

Such inconsistency occurs when the particular combination of labels in the merged graph violates constraints from the model fragment's metamodel, from some design pattern that the software process requires to follow, etc. Since a precise taxonomy of errors is intimately linked to the concepts of automatic merge and diff between two model fragments, views or artifacts, it will be presented together with the definition of these related concepts in the next deliverable D2.2 that focuses on these issues.

In the next paragraph, we explain the semantics of each class in the diagram of Figure 1 in one subsection.

3.1 CLASS NAMED ENTITY

An abstract class generalizing all classes with the attribute name, *i.e., Project, Participant, HistoryUnit, AfDiff and AfAction.*

3.2 CLASS GALAXY FRAMEWORK

3.2.1 Definition

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

An abstract object-oriented framework of which concrete instantiations provide revision control services for large-scale collaborative MDE projects. The collaborative unit defined in

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

Figure 1 is the main concrete class of the Galaxy framework. As shown in Figure 8, a Galaxy framework assembles other classes beyond the collaborative units, notably the revision strategy. A revision control system for MDE based on the Galaxy project implements a given Galaxy framework instance. All such instances share instances of the same general-purpose concrete collaborative unit class. In contrast, each specific instance of the framework assembles instances of a specific concrete specialization of the abstract class revision strategy.Therefore , a particular Galaxy revision systems results from assembling collaborative units with a particular revision strategy. This way, the revision strategy can be customized to the specific revision control needs of a specific class of projects in order to ensure the scalability of the revision control services provided by the Galaxy framework instantiation for those projects.

3.2.2 Properties

- **pj:** the projects under revision control using a Galaxy framework instance;
- **pa**: the participants of the projects under revision control using a Galaxy framework instance;
- **url**: the web address to access the administration services of a Galaxy framework instance.

3.3 CLASS PROJECT

3.3.1 Definition

An MDE project under revision control using a galaxy framework instance.

3.3.2 Properties:

- **gf:** the galaxy framework instance providing revision control for the projet
- **creator:** the galaxy framework user who created the project;
- **pa:** the participants of the project.
- **client:** an MDE project that is reusing a model built in another MDE project as model component;
- **server:** an MDE project providing a model component for other MDE projects.

3.4 CLASS PARTICIPANT

3.4.1 Definition

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

A participant to an MDE project under revision control using a galaxy framework instance.

3.4.2 Properties:

- **gf:** the galaxy framework instance providing revision control for the projects to which the participant takes part;
- **created:** the projects created by the participant;
- **pj:** the projects to which the participant takes part through the galaxy framework instance;
- **co:** the trace of the commit actions performed by the participant;
- **tg:** the tags put by the participant on commit objects;
- **lt**: the locks on commit objects that the participant currently holds.

3.5 ASSOCIATION CLASS COLLAB UNIT

3.5.1 Definition

A local workspace owned by one project participant storing the artifacts of one collaborative MDE project relevant to the participant. We make the simplifying assumption that if a participant takes part to multiple projects using the same galaxy framework instance for revision control, it owns one distinct collaborative unit per project to which (s)he takes part. The notion of collaborative unit is purely conceptual. It is thus neutral with respect to the specific collaborative workflows, artifact persistence technology and its deployment over a network. Consequently, is can model either a local workspace in a centralized RCS such as svn, or a local repository in a distributed RCS such as git, hg or bzr.

3.5.2 Properties:

- **url**: the web address that serves as entry point to the galaxy;
- **blessed:** a boolean indicating that the collaborative unit plays the role of central reference repository in a centralized copy-revise-merge collaborative workflow; when this is the case, all non-blessed collaborative units synchronize their respective revision only with a single blessed collaborative unit and not directly among themselves; a separate synchronization network among several blessed collaborative units can also be independently set up to insure the redundancy needed for continued and responsive service in the occurrence of node and/or connection downtime;

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **pj:** the project for which the collaborative unit was created;
- **pa:** the galaxy framework instance user owning the collaborative unit;
- **pusher:** the set of remote collaborative units allowed to send artifact change notifications to the host collaborative unit (using the *publish* operation);
- **pushee:** the set of remote collaborative units to which the host collaborative unit is allowed to send artifact change notification (using the *publish* operation);
- **puller:** the set of remote collaborative units allowed to pull updates of their artifacts from those stored in the host collaborative unit (using the *update* operation);
- **pullee:** the set of remote collaborative units from which the host collaborative unit is allowed pull updates of its artifacts (using the *update* operation);
- **origin:** the remote collaborative unit from which the initial revision of the artifacts were copied into the host collaborative unit artifacts were (using the *clone* operation); becomes the default source for subsequent *update* operations;
- **localBt:** a set of pointers to the development branches whose artifacts might have been locally changed since their last updates from a remote collaborative unit;
- **localMain:** a pointer, member of *localBt*, to the branch that holds the main development trunk of the project;
- **head:** a pointer, member of *localBt*, to the branch containing the artifacts that have been checked out for edition by the client application of the host collaborative unit;
- **remoteBt:** a set of pointers to untouched copies of the branches pointed to by *localBt* as they were at the time of their last update from a remote collaborative unit;
- **remoteMain:** a pointer to an untouched copy of the main development branch as it was when it was last updated from a remote collaborative unit.

3.5.3 Components

Class *HistoryUnit* (see section 3.6). The information units of a collaborative unit are the artifacts of project pj relevant for participant pa, together with the meta-data used to control their revision.

3.5.4 Operations

3.5.4.1 clone(afId: String, fromCuUrl: String, atTagId: String, atRev: Integer)

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

Copies to the host collaborative unit a copy of the artifact which *uuid = afId*, pointed to by the commit object with revision number *atRev* and tagged by the tag which *uuid = atTagId* inside the remote collaborative unit located at the web address *fromCuUrl*. It addition to the artifact itself, it also copies the versioning meta-data about the artifact stored in the above mentioned commit and tag objects; the remote collaborative unit then fills the host collaborative unit's origin property; This operation is allowed only if the host collaborative unit has been previously added to the puller property of the remote collaborative unit which *url = fromCuUrl*. Clone is realized by calls to the operation of the same name on the artifact objects and artifact revision meta-data objects (instances of classes HistoryUnit and Diff) contained by the host collaborative unit.

3.5.4.2 commitl(afId: String, msg:String, afas: AfAction[*]): Commit

Commits to the local collaborative unit *cu* the new revision of the artifact *af* with *uuid = afId. afas* corresponds to the canonical sequence of artifact revision actions (defined in section 3.13) needed to apply to the latest revision of one stored in *cu*[4]obtain *af. Commitl* first creates a copy of *af*. It then creates a new commit object *co* pointing to this new copy. It then makes the *head* branch points to *co* as the updated *tip* of its revision history. It returns *co*.

3.5.4.3 commitg(afId: String, msg:String, afas: AfAction[*], fromPaUrl: String, toCuUrl: String): Error[*]

Commits to a blessed collaborative unit *bcu* with *url = toCuUrl* the new revision of the artifact *af* with *uuid = afId. Commitg* first checks whether the host collaborative unit's *participant* is the *author* of the last *commit* for *af* on *bcu*.. If it is the case, *commitg* first creates in *bcu* a copy of *af*. It then creates in *bcu* a new commit object *co* pointing to this new copy. It then makes the *head* branch in *bcu* point to *co* as the updated *tip* of its revision history and returns an empty error list. If the *participant pa1* who last committed a revision *af1* of *af* to *bcu* is not the host collaborative unit's *participant, commitg* it then calls *merge* taking as arguments *afId* and the *head* branch id at *bcu*..

3.5.4.4 merge(afId: String, withBtId: String): Error[*]

---

[4] i.e., the one pointed to by the *tip* commit object of the host collaborative unit's *head* branch.

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

Attempts to automatically merge the latest revision of artifact which *uuid = afId* in the head branch of the host collaborative unit, with the revision of the same artifact in the branch which *uuid = withBtId*. If they are *not* automatically mergeable, it leaves the two revisions unchanged and returns the conflicts that caused the merge failure. If the two revisions are mergeable, it then calls the *audit* operation on the artifact resulting from the successful merge. If the audit reports at least one inconsistency, it leaves both revisions of the input artifact unchanged, and returns the inconsistencies returned by the audit. If the audit reports no inconsistency, it creates a new commit object *co* having two fillers for its previous property: the commit node at the tip of the head branch before the call to merge, and the commit node at the tip of the branch with *uuid = withBtId*. It then fills the tip property of the head branch with *co*. This general merge operation on two arbitrary artifacts is realized by calls to specific merge operations on product units, reuse units, method units, views, model elements, model element references and model element attributes. As shown in Figure 8, these specific operations are abstract. It is their specializations in concrete operations with the same signature that define the merge strategy of a specific galaxy framework instance.

3.5.4.5 audit(af: Artifact): Inconsistency[*]

Audits artifact which uuid = afId and returns the inconsistencies that it contains (if any).

3.5.4.6 update(afId: String, fromCuUrl: String, atTagId: String, atRev: Integer): Error[*]

Firts fetches the revision of artifact which *uuid = afId* pointed to by the commit object of revision number *atRev* and tagged by the tag which *uuid = atTagId* in the remote collaborative unit located at web address *fromCuUrl*. The calls merge with afId as afId paramenter and atTagId as btId parameter.

3.5.4.7 revert(afId: String, toRev: Integer)

Backtracks to revision number *toRev* of the artifact which *uuid = afId*. Just changes the current branch *tip* so that it points to the commit object of revision number *toRev*.

3.5.4.8 tag(afId: String, tagName: String, msg: String, atRev: Integer): Tag

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

Creates in the host collaborative unit a new tag object named tagName with message msg to the commit object with revision number atRev for artifact which uuid = afId. Adds only a simple tag, such as a release tag. Does not work properly for branch and lock tags.

3.5.4.9 renameTag(afId: String, tagId: String, newTagName: String)

Changes to *newTagName*, the name of the tag which *uuid = tagId* for artifact which *uuid = afId*.

3.5.4.10 addBranch(afId: String, btName: String, msg: String): BranchTag

Creates in the host collaborative unit a new development branch named *btName* with message msg for the artifact which *uuid = afId*. Creates a new branch tag object which *tip* points to the commit object already pointed to by the current branch;

3.5.4.11 delBranch(afId: String, btId: String)

Deletes from the host collaborative unit the branch tag object which *uuid = btId* for artifact which *uuid = afId*. Depending on the memory management policy adopted may also trigger deletion of all the objects accessible from the *tip* property of the deleted branch object by recursively following the *previous, af, diff* and *ea* properties of commit objects.

3.5.4.12 switchTo(afId: String, btId: String)

Switches the head property of the host collaborative from its current value to the branch object which *uuid = btId* for artifact which *uuid = afId*.

3.5.4.13 rebase(afId: String, btId: String, fromBtId: String, ontoBtId: String): BranchTag

Linearizes three branches for artifact which *uuid = afId* in the host collaborative unit into two in order to make past development history easier to follow. First finds the commit object *coBase* from which the branch which *uuid = btId* diverged from the branch which *btId = fromBtId*. Then replays the artiact actions associated to all commit objects between *coBase* and the *tip* of *btName* onto the *tip* of a third branch which uuid = *ontoBtId*. Finally deletes the branch *btId* which is then no longer needed. Repeated calls to this rebase operation allows to fully linearize a very branchy, hard to follow development history with a lot of trials and errors into a simpler, neat, linear one.

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

Introduced by Git, this rebasing concept has since then been adopted by other DRCS such as Mercurial and Bazaar.

3.5.4.14 publish(co: Commit, toCuUrl: String)

Notifies the remote collaborative unit at web address *toCuUrl* that a new commit object *co* has been added to the host collaborative unit. Requires the target remote collaborative unit to be registered as *pushee* of the host collaborative unit.

3.5.4.15 publishTag(tg: Tag, toCuUrl: String)

Notifies the remote collaborative unit at web address *toCuUrl* that a new tag object *tg* has been added to the host collaborative unit. Requires the target remote collaborative unit to be registered as *pushee* of the host collaborative unit.

3.5.4.16 publishRenameTag(afId : String, tagId : String, newTagName : String, toCuUrl : String)

Notifies the remote collaborative unit at web address *toCuUrl* that the tag object *which uuid = tagId* had been renamed *newTagName* in in the host collaborative unit. It requires the target remote collaborative unit to be registered as *pushee* of the host collaborative unit.

3.5.4.17 publishDelTag(afId: String, tagId: String, toCuUrl: String)

Notifies the remote collaborative unit at web address *toCuUrl* that the tag which *uuid = tagId* has been deleted from the host collaborative unit. It requires the target remote collaborative unit to be registered as *pushee* of the host collaborative unit.

3.5.4.18 lock(afId: String, atBtId: String, forCuUrl: String): LockTag

Locks the artifact which *uuid = afId* at the tip of the branch which *uuid = atBtId,* for exclusive changes by participant whose collaborative unit web address is *forCuUrl.* First creates a lock tag object *lt* in host collaborative unit that points the latest commit object for the artifact. In a centralized setup, lock then calls *publishTag* with *lt* as parameter to notify the single blessed collaborative unit of the lock. In a decentralized setup, lock then notifies all the collaborative units registered as *pushees* of the host collaborative unit of the lock. Upon reception of this lock

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

notification, these pushees in turn notify their pushee, thus triggering a recursive propagation of the lock throughout the whole galaxy. This distributed version works properly only if all the galaxy collaborative units for a given project are reachable from any other galaxy collaborative unit for that project through pushee, pusher relationships. Calling lock is only permitted when the commit object which points to *af* in the host collaborative unit it itself free of lock tag.

3.5.4.19 unlock(afId: String, atBtId: String)

Unlocks the artifact *af* which *uuid* = *afId* at the tip of the branch which *uuid* = *atBtId*. First deletes the lock tag in the host collaborative unit that points to the latest commit object for the artifact. In a centralized setup, lock then calls *publishDelTag* with *lt* as parameter to notify the single blessed collaborative unit of the lock release. In a decentralized setup, lock then notifies all the collaborative units registered as *pushees* of the host collaborative unit of the lock release. Upon reception of this lock release notification, these pushees in turn notify their pushee, thus triggering a recursive propagation of the lock release throughout the whole galaxy. This distributed version works properly only if all the galaxy collaborative units for a given project are reachable from any other galaxy collaborative unit for that project through pushee, pusher relationships. Calling unlock is only permitted when there is a lock tag on the commit object pointing to *af* in the host collaborative unit.

3.6 ABSTRACT CLASS HISTORY UNIT

3.6.1 Definition

An abstract class that generalizes the three main classes of objects used to store artifact revision history in a collaborative unit (1) Revision unit for project data, and (2) Commit and Tag, for revision control metadata.

3.6.2 Properties

- **uuid:** a string allowing to identifying uniquely all the revisions of the same *HistoryUnit* objects in distinct collaborative units of the galaxy, even if they differ in terms of content;

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **hash:** a string allowing rapid indexing and (almost) uniquely identifying *HistoryUnit* objects in the whole galaxy *by their content;*
- **size:** number of memory bytes occupied by the HistoryUnit object;

3.7 ABSTRACT CLASS REVISION UNIT

3.7.1 Definition

An abstract class that generalizes all project data under revision control (versioning). Our first general assumption is that a modeling team may wish to version model elements and model views. This is why *RevisionUnit* generalizes the concrete classes *ModelElt* and *View*. Exactly which class of model elements and model view should be versioned depends on a specific modeling language, MDE process and CASE tool connected to an instance of the galaxy framework. It is during the instantiation of the framework that *RevisionUnit* can be constrained to generalize not all instances of *ModelElt* and *View* but only restricted subclasses of them. model elements, model views and the artifacts storing them in collaborative units (see section 4). The revision number of model elements and views are derived from the artifacts containing them that are persistently stored in collaborative units and exchanged among them for synchronization. In turn, the revision number of any such artifact (project data) is derived from the revision number of the commit object (project history metadata) that is created when the artifact is committed to the collaborative unit.

3.7.2 Properties

- **uuid:** a string allowing to identifying uniquely all the revisions of the same RevisionUnit objects in distinct collaborative units of the galaxy, even if they differ in terms of slots;
- **hash:** a string allowing rapid indexing and (almost) uniquely identifying *RevisionUnit* objects in the whole galaxy *by their slot content.*
- **mutable:** a boolean indicating whether the artifact is read-only or can be altered; in the MDE context, example of non-mutable artifacts, from the perspective of an application developer, include the classes forming the meta-model of the model under construction, the built-in architectural framework that this model specializes or the built-in types provided by the modeling language;

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **locallyChanged:** a boolean that is true iff the revision unit has been locally modified since it was last committed (either locally using *commitl*, or globally using *commitg*) or updated;
- **remotedChanged:** a boolean that is true iff the revision unit has been modified in a remote collaborative unit, since it was last committed in, committed to or updated to the local collaborative unit; the local collaborative can know about such change only if it was carried out in a remote collaborative unit of which it is a puller; in centralized mode, all collaborative units are puller of a blessed collaborative unit;
- **conflicted:** a boolean which is true iff the last update attempt from a remote collaborative resulted in at least one conflict;
- **published:** a boolean which is true iff the artifact has been published by the owner of the collaborative unit where it is stored, so that other project participants can update their own copy of this artifact in their collaborative units with this more recent revision; committing an artifact while not publishing it allows an artifact to be persistent while remaining private to a given collaborative unit;

3.8 ABSTRACT CLASS ARTIFACT

3.8.1 Definition

Any artifact constructed during a software project, whether it be a natural language document, a code file or a model that follows a meta-model. Due to its genericity, we model this concept as an abstract class.

3.8.2 Properties

- **url:** the web address where the artifact is stored;
- **localPath:** the tail of the url that starts after the url of the collaborative unit where the artifact is stored;
- **prevRev:** pointers to the previous local revisions of the artifact in the collaborative unit (if any);
- **nextRev:** pointer to the next local revision of the artifact in the collaborative unit (if any);
- **co:** the commit object that points to the artifact;
- **fromRev**: the diff between the artifact and its previous revision (if any);
- **toRev**: the diff between the artifact and its next revision (if any).

3.8.3 Operations

- **diff(withAfid: String): AfDiff[*]**: returns all the differences between the host artifact and the argument artifact; an abstract operation that must be specialized into a concrete one, for each different concrete

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

subclass of the abstract class artifact; these specializations are beyond the scope of D2.1 and will be defined in the D2.2;

- **clone(): Artifact:** returns a copy of the host artifact; an abstract operation that must be specialized into a concrete one, for each different concrete subclass of the abstract class artifact;

3.9 ABSTRACT CLASS ATOMIC ARTIFACT

An artifact that does not contain any other artifact. A leaf in the artifact containment tree.

3.10 ABSTRACT CLASS COMPOSITE ARTIFACT

3.10.1 Definition

An abstract subclass of *Artifact* that contains other artifacts. Form a complete and disjoint generalization set of Artifact with *AtomicArtifact*. Together, these three classes follow a composition pattern that models a containment tree. That they are structured in such tree is the only assumption made about artifacts at the level of the collaborative unit. This design choice is motivated by the pervasiveness of such structure in MDE CASE tools, file systems and XML documents. In an MDE CASE tool, model elements are structured in such tree following the composition meta-associations of the meta-model. The nesting of elements in XML documents also follows this containment pattern. File systems are similarly structured with the individual files being the atomic artifacts and the folders being the composite artifacts.

3.10.2 Properties

- **ls:** a pointer to the artifacts *directly* contained by the composite artifact (*i.e.*, nested at level 1);
- **lsr:** a pointer to *all* the artifacts nested in the composite artifact, whether contained at its top-level (i.e., nested at level 1) or recursively contained in one of its nested composite artifacts (i.e., nested at any level > 1);

3.10.3 Operations

- **add(afId: String):** nests a new artifact of *uuid* = *afId* inside the composite artifact;
- **del(afId: String):** deletes the artifact of *uuid* = *afId* from the composite artifact;

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **move(afId: String, fromPath: String, toPath: String):** moves nested artifact of *uuid* = *afId* inside the host composite artifact containment tree from being a direct child of nested composite artifact of *uuid* = *fromPath* to being a direct child of nested composite artifact of *uuid* = *toPath*;

3.11 CLASS COMMIT

3.11.1 Definition

A snapshot node, in the revision history, of an artifact stored in a collaborative unit. One object of this class is added to the revision history each time a new version of the artifact is committed to one collaborative unit (by executing the operation commit).

3.11.2 Properties

- **author:** the participant that executed the commit action that the commit object records in the revision history;
- **timeStamp:** the date and time when the commit action was executed;
- **msg:** a text that describes what has been changed from the last revision of the committed artifact and explains the motivation for the change;
- **revision:** an integer which gets incremented every time the artifact is committed;
- **previous:** a pointer to the previous commit object for the same artifact (if any);
- **next:** pointer(s) to the next commit object for the same artifact (if any); there can be several ones if the commit object served as base for a branch;
- **author:** the project participant who called the commit operation which resulted in the creation of the commit object;
- **tg:** the tag that points to the commit object;
- **bt:** the subset of tg that contains only the branch tags;
- **af:** a pointer to the persistent snapshot of the new revision of the committed artifact;
- **diff:** a pointer to an *AfDiff* object containing all the differences between *af* and its revision and
- *af. prevRev*, its preceding revision in the revision history;
- **ea:** a pointer to the canonical sequence of artifact actions whose execution would produce *af* from *af.prevRev*.

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

Taken together, the *af*, *diff* and *ea* properties allows a collaborative unit to implement any representation used for commit nodes in RCS such as svn, git, hg and bzr, and in distributed CASE tools with trivial revision policies such as Praxis [4], [15]. Maximizing scalability, involves devising clever combination of these three representations.

Note that a galaxy commit object does not necessarily points to the root of the entire artifact tree as do commit objects in state-of-the-art DRCS git, hg and bzr. This is because the revision operations of a galaxy collaborative unit can be executed any sub-artifact nested down the artifact containment tree. This feature is not currently supported by the consolidated versions of DRCS which revision always apply to the full versioned artifact containment tree of the entire project. However, there is an experimental git library that addresses this need for partial cloning, checkout, commit and merge. But it has not yet been tested on large projects. Revision operations at the low-level grain of individual artifacts seem a valuable option to limit space requirement of non-blessed collaborative units as well as avoiding updating locally irrelevant artifacts. However, how we will illustrate in section 7.4, it also makes the automated avoidance of dangling references among artifacts most complex and thus potentially computationally costlier. This suggests that there exists some fundamental trade-off between artifact grained and collaborative unit grained revision operations. Supporting any grain is thus an important feature of collaborative unit concept proposed for Galaxy. Note that to implement a policy similar to that of current DRCS using a galaxy collaborative unit is simply a matter of using it with a process that forces all revision operations to be called using as first argument the top-level node in the artifact containment tree.

3.11.3 Operations

- **clone():Commit,** creates a copy of the *Commit* object;

3.12 ABSTRACT CLASS AFDIFF

3.12.1 Definition

An abstract class representing the differences between two revisions of the same artifact

3.12.2 Properties

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **self:** the host artifact object that called the diff operation;
- **with:** the artifact with which the host compared itself by calling such an operation;
- **base:** the common ancestor (if any) revision in the artifact's history from which *self* and *with* originated before diverging is separate change branches;
- **co:** the commit node (if any) that points to the *AfDiff* object as an alternative persistent storage format to the whole snapshot of the artifact revision it committed;
- **base2Self**: the canonical action sequence which results into *self* when applied to *base;*
- **base2With**: the canonical action sequence which results into *with* when applied to *base;*
- **self2With**: the canonical action sequence which results into *with* when applied to *self;*
- **with2Self**: the canonical action sequence which results into *with* when applied to *base;*

3.12.3 Operations

- **clone():AfDiff**, creates a copy of the *AfDiff* object;

3.13 ABSTRACT CLASS AFACTION

3.13.1 Definition

An abstract class modeling the minimal set of primitive change actions on artifacts into which any artifact manipulation can be ultimately decomposed. The concrete classes specializing AfAction include the addition, deletion and move of product units, reuse units and method units inside their container product unit. They are defined in section 5.

3.13.2 Operations

- **clone():AfAction,** creates a copy of the *AfAction* object;

3.14 CLASS TAG

3.14.1 Definition

A label put on a *Commit* object in the revision history of an artifact. Direct instances of this concrete class can be leveraged for a variety of purposes, notably the fast retrieval of release versions.

3.14.2 Properties

- **co:** a pointer to the *Commit* object being tagged;
- **author:** the participant that put the tag on the co commit object;

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **timeStamp:** the date and time when the tag was created;
- **msg:** a text giving the semantics of the tag.

3.14.3 Operations

- **clone():Tag,** creates a copy of the *Tag* object;

3.15 CLASS BRANCH TAG

3.15.1 Definition

A special kind of tag indicating a development branch. This *branching as tagging* approach pioneered by git is in sharp contrast with the *branching as copy* approach used by svn. It was one of the most dramatic scalability improvement brought about by git.

3.15.2 Properties

- **tip:** a pointer to the *Commit* object that in turns points to the lastest revision of the artifact in the host branch.
- **lt:** the lock tag that currently locks the branch;

3.15.3 Operations

- **clone():BrancgTag,** creates a copy of the *BranchTag* object;

3.16 CLASS LOCK TAG

3.16.1 Definition

A special kind of tag used to lock an artifact for the exclusive, non-concurrent edition of the artifact revision accessible from the *Commit* object to which the *LockTag* object points to. This design allows reusing the *delTag, publishTag, publishDelTag* operations to propagate locks and lock releases among collaborative units.

3.16.2 Properties

- **bt:** a pointer to the branch that is locked;
- **for**: the project participant currently holding the lock on *bt*

3.16.3 Operations

- **clone():LockTag,** creates a copy of the *LockTag* object;

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

4. DEFINITION OF VIEWS AND MODEL FRAGMENTATION MECHANISMS

In the previous section we focused on the concern of defining meta-data to support revision control for software artifact of *within any software engineering paradigm*. In this section, we specialize the abstract class Artifact defined in the previous section into concrete classes to address the central issue that is specific to revision control *within the MDE paradigm*: how to group model elements in very large models into *intermediate grain structures* that can be scalably manipulated for distributed revision control purposes? By "intermediate grain" here we mean any grain between the coarser possible grain of the whole project megamodel and the finest possible grain of individual model elements.

4.1 HOW TO STRUCTURE MODEL ELEMENTS AND VIEWS ON THEM FOR REVISION CONTROL PURPOSES?

The intermediate grains that we propose are derived from the following reasoning. First, we distinguish between three general classes of intermediate grain structures: (1) flat model partition, (2) flat model element overlapping subsets and (3) nesting containment tree. They are respectively illustrated on the left part of Figure 2, on the right part of Figure 2 and in Figure 3. In these figures, individual model elements are represented by small grey squares, references between them by lines and model fragments of intermediate grains as large white squares or ellipses. Then, we notice that we already used the containment tree structure for the abstract artifact concept that relates the paradigm-independent collaborative unit model presented in the previous section.

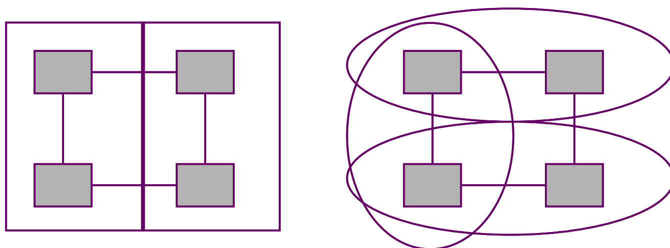

Figure 2: Flat model fragmentation: partition *vs.* overlapping model element subsets

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

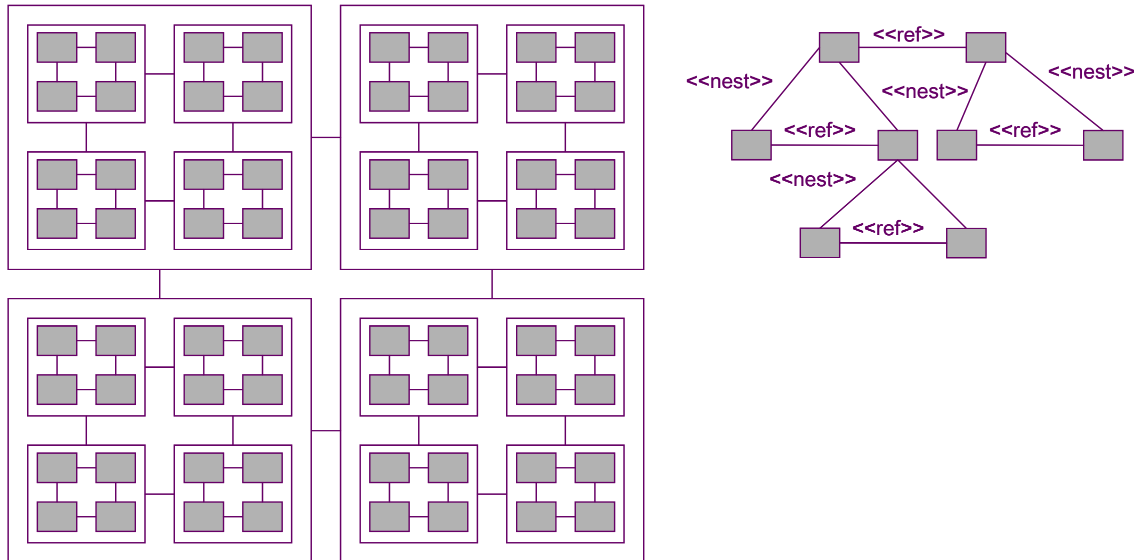

Figure 3: Model fragmentation as a containment tree with references

In Figure 3, we do not show any direct reference between elements at different depths in the containment tree because such cross-level references can always be realized by a path constituted of intra-level references and nesting relationships between an element and its parent or child in the containment tree.

Then, we notice that the three intermediate structures of Figure 2 and in Figure 3, far from being mutually exclusive, can be combined to maximize representational flexibility. We also notice that the decomposition of a system in subsystems, components, packages or other *reuse unit*s, aims at structuring the system into highly cohesive units that are very loosely coupled between them. Such units are thus near partitions. Within the MDE paradigm, these reuse units can thus be considered to partition the model elements of a software model, even though there exist references between elements in different partition set. In contrast, views, aspects, diagrams and other software *method units* significantly overlap between themselves while cutting across *reuse unit* boundaries.

These observations lead us to propose the MDE artifact model of Figure 4. In this model the paradigm-independent abstract class *CompositeArtefact* from the collaborative unit model is

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

specialized into the *ProductUnit* concrete class. This specialization restricts the nested artifacts of a product unit to be either product units, reuse units or method units. This sub-categorization is mutually exclusive and covers all cases (this is represented in Figure 4 by the OCL invariant attached to ProductUnit). Reuse units and method units are concrete classes that form a mutually exclusive and covering sub-categorizations of the abstract class *AtomicArtifact* from the collaborative unit model. Atomic artifacts are atomic in the sense of not containing any other *artifact*. However reuse units contain model elements and method units contain views. There is no contradiction here since model elements are *not* artifacts.

This design choice allows full decoupling between, on the one hand, the direct structural relationships among elements *in the model*, and one the other hand, the structural relationships among meta-data used to support revision control of the model. It thus insures the versatility of the key concepts of **product unit, reuse unit** and **method unit** for a variety of software modeling languages, methods, processes and application domains. In section 6.7 we also introduce the concept of a **revision strategy** defined as a *mapping* from (a) classes of model elements onto product units and reuse units and (b) classes of model views onto product units and method units. A key step in the process of instantiating the general galaxy RCS framework presented in this document into a concrete DRCS service to which MDE CASE tools can delegate model revision control is to define the revision strategy that the concrete DRCS realizes. Note that what we call model element classes for revision strategy definition *can but do not necessarily* correspond to metaclasses in the metamodel(s) of the modeling language(s) used in the model. They could be for instance model elements that contained elements that together occupy some space threshold.

Similarly, classes of views for revision strategy definition *can but do not necessarily* correspond to the viewpoints or diagram classes defined used in the model.

Having explained the general design rational behind the model of Figure 4, we now proceed to give precise definition for each of its elements.

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

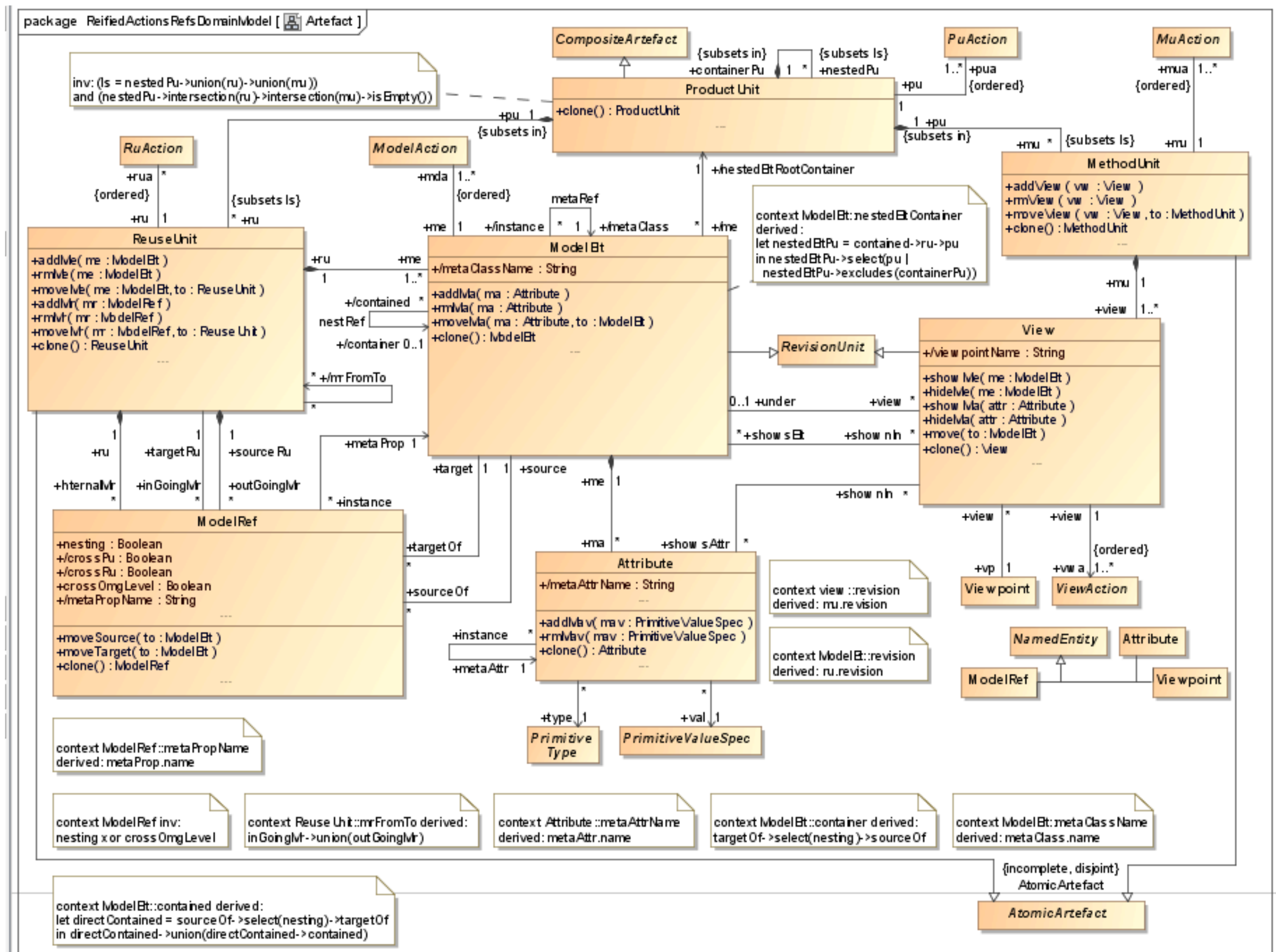

Figure 4: Concepts to structure models and views for revision purposes

4.2 CLASS MODEL ELT

4.2.1 Definition

A model element.

4.2.2 Properties

- **uuid:** string that uniquely identifies several model element with distinct content contained in different galaxy artifacts as being different revisions of the same reference
- **metaClass:** a property pointing to the metaClass of which the model element is an instance; it is derived through a reference between the model element of the model and its metaclass as a model element of the metamodel;

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **metaClassName:** a derived property containing the name of the model element's metaclass
- **mda:** the construction trace of the model element in terms of model actions.
- **ma:** the attributes of the model elements;
- **contained:** the model elements contained in the model element; an element W contains an element P iff there is nesting model reference from W to P;
- **container:** the model element that contains the model element;

4.2.3 Operations

- **addMa(ma: Attribute):** adds attribute *ma* to host model;
- **rmMa(ma: Attribute):** removes attribute *ma* from host model;
- **moveMa(ma: Attribute, to: ModelElt):** moves attribute *ma* from host model element to the *to* model element;
- **clone():ModelElt,** creates a copy of the host *ModelElt* object;
- **diff(withMeId: String): MeDiff,** returns all the differences between the host model element and the model element which uuid = withMeId. The details on how to realize this operations and the structure of its results will be given in D2.2.

4.3 CLASS ATTRIBUTE

4.3.1 Definition

A model element property typed by a primitive type such as a MOF, OCL, Ecore or XML type: boolean, integer, real, string, date, time, dateTime, uri or collections of such types.

4.3.2 Properties

- **metaAttr:** the property of the meta-model that the attribute , (*i.e.,* from OMG level N down to OMG level N-1) instantiates;
- **metaAttrName:** a derived property of the attribute containing the name of property of the attribute's meta-attribute.
- **me:** the model element that contains the attribute.

4.3.3 Operations

- **addMav(mav:PrimitiveValueSpec,):** adds value *mav* to host attribute *ma*;

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **rmMav(mav: PrimitiveValueSpec,):** removes value *mav* to host attribute;
- **clone():Attribute,** creates a copy of the host *Attribute* object;
- **diff(withMaId: String): MeDiff,** returns all the differences between the host attribute and the attribute which uuid = withMaId. The details on how to realize this operations and the structure of its results will be given in D2.2.

4.4 CLASS MODEL REF

4.4.1 Definition

A model element property typed by another model element.

4.4.2 Properties

- **source:** the model element source of the reference;
- **target**: the model element target of the reference;
- **nesting**: a boolean which is true iff the reference models a model element containment relationship in which the source is the container and the target the contained element;
- **crossPu**: a boolean which is true iff its source and target model elements are contained in different product units, (derived property)
- **crossRu**: a boolean which is true iff its source and target model elements are contained in different reuse units (derived property)
- **crossOmgLevel**: a boolean which is true iff the reference's source and target pertain to different OMG modeling level, for example a reference from model element (level 1) to a meta-model element (level 2) or from a meta-model element to a meta-meta-model element (level 3);
- **metaProp**: the property of the meta-model of which the reference is a instance;
- **metaPropName**: a derived attribute containing the value of the name property of the reference's metaProp property.
- **ru:** the reuse unit to which the model reference pertains; a model reference is also stored in the reuse unit that stores its source model element;
- **sourceRu**: the reuse unit containing the source element of the model reference;
- **targetRu**: the resue unit containing the target element of the model reference;

4.4.3 Operations

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **moveSource(to:** ModelElt**):** change source property of host reference to *to* model element;
- **moveTarget(to:**ModelElt): change target property of host reference to *to* model element;
- **clone():Attribute,** creates a copy of the host *ModelRef* object;
- **diff(withMrId: String): MrDiff,** returns all the differences between the host model reference and the model reference which uuid = withMrId. The details on how to realize this operations and the structure of its results will be given in D2.2.

4.5 CLASS VIEW

4.5.1 Definition

A methodologically defined partial model fragment showing the model elements, references and attributes relevant to a single concern; may represent views in orthographic MDE [2] aspects in Aspect-Oriented Modeling (AOM), diagram in UML modeling, etc.

4.5.2 Properties

- **under:** the model element, if any, of which the view is a partial fragment;
- **showsElt:** the model elements shown in the view;
- **showsAttr:** the model element attributes shown in the view;
- **mu:** the method unit containing the revision control meta-data for the view;
- **vp:** the viewpoint defining the schema of the view, *i.e.,* constraints defining what classes of model element, model element references and model element attributes are allowed to appear in the view;
- **viewpointName:** a derived property containing the name of the viewpoint;
- **vwa:** view construction trace as a sequence of show and hide actions on model elements, model element references and model element attributes.

4.5.3 Operations

- **showMe(me: ModelElt):** shows model element *me* in the view of the host[5] method unit;
- **hideMe(me: ModelElt):** hides model element *me* from the view of the host method unit;
- **showMr(mr: ModelRef):** shows model element reference *mr* in the view of the host method unit;
- **hideMr(mr: ModelRef):** hides model element reference *mr* in the view of the host method unit;

---

[5] In this document we call "host" the object which operation is invoked.

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **showMa(ma: Attribute):** shows model element attribute *ma* in the view of the host method unit;
- **hideMa(ma: Attribute):** hides model element attribute *ma* in the view of the host method unit;
- **move(to: ModelElt):** moves the view from under the model element under to the model element parameter of move;
- **clone():** View, creates a copy of the *View* object;
- **diff(withViewId: String): ViewDiff,** returns all the differences between the host model view and the model view which uuid = withViewId. The details on how to realize this operations and the structure of its results will be given in D2.2

4.6 CLASS METHOD UNIT

4.6.1 Definition

A container of model views for scalable revision control purpose.

4.6.2 Properties

- **pu**: the product unit containing the method unit;
- **view**: the view which revisions are controlled by the method unit;
- **mua**: the construction trace of view as a sequence of method unit operation calls;

4.6.3 Operations

- **addView(vw: View):** adds view *vw* to the host method unit;
- **rmView(vw: View):** removes view *vw* to the host method unit;
- **moveView(vw: View):** moves view *vw* from the host method unit to the method unit *to*.
- **clone(): MethodUnit,** creates a copy of the host *MethodUnit* object;
- **diff(withMuId: String): MuDiff,** returns all the differences between the host method unit and the method unit which *uuid* = *withMuId*.

4.7 CLASS REUSE UNIT

4.7.1 Definition

A container of model elements for scalable revision control purpose. What classes of model element it can contain is defined by a model fragmentation strategy as defined in section 6.

4.7.2 Properties

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **pu**: its product unit container;
- **rua**: the construction trace of the reuse unit as a reuse unit action sequence;
- **mr**: the model references contained in the reuse unit;
- **me:** the model elements contained in the reuse unit.
- **mrFromTo:** reuse units that contain model references which source or target elements are in the reuse unit;

4.7.3 Operations

- **addMe(me: ModelElt):** adds model element *me* to the host reuse unit;
- **rmMe(me: ModelElt):** removes model element *me* from the host reuse unit;
- **moveMe(me: ModelElt, to: ReuseUnit):** moves model element *me* from the host reuse unit to the *to* reuse unit;
- **addMr(mr: ModelRef):** adds model element reference *mr* to the host reuse unit;
- **rmMr(mr: ModelRef):** removes model element reference *mr* from the host reuse unit;
- **moveMr(mr: ModelRef, to:ReuseUnit):** moves model element reference *mr* from the host reuse unit to the *to* reuse unit;
- **clone(): ReuseUnit,** creates a copy of host reuse unit with the same properties;
- **diff(withRuId: String): RuDiff,** returns all the differences between the host reuse unit and the reuse unit *which uuid = withRuId.*

4.8 CLASS PRODUCT UNIT

4.8.1 Definition

Software project structuring unit of larger grain than the minimal reuse units and methodological views for revision control purposes. Allows defining structures of arbitrary grain through a recursive containment relationship. A model fragmentation strategy maps specific classes of composite model elements or methodological groups of such elements onto product units.

4.8.2 Components

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

- **nestedPu:** product units directly contained in the host product unit, i.e., its child in the product unit containment tree;
- **ru:** reuse units contained in the host product unit;
- **mu**: method units contained in the host product unit.

4.8.3 Properties

- **containerPu:** product unit that directly contains the host product unit, i.e., its parent in the product containment tree;

4.8.4 Operations

- **clone():Attribute,** creates a copy of the host *ProductUnit* object;
- **diff(withPuId: String): PuDiff,** returns all the differences between the host product unit and the product unit which *uuid* = *withPuId*. The details on how to realize this operations and the structure of its results will be given in D2.2.
- To add to, remove from and move product units, reuse units and method units inside a product unit, one can use the operations add, del and move that *ProductUnit* inherits from it superclass *CompositeArtifact* (shown in Figure 1).

5. ACTIONS TO EXECUTE DURING THE COLLABORATIVE UNIT LIFE CYCLE

In this section, we define two distinct, complementary minimal sets of primitive revision actions into which higher-level ones can all be ultimately decomposed. The first set, shown in Figure 5, specialize the abstract class VuAction. They correspond to the primitive model and view edition actions executed by a CASE tool in response to user actions with its GUI. The second set, shown in Figure 6, specialize the abstract class AfAction introduced in section 3.13. They correspond to primitive artifact revision actions executed by an instance of the galaxy framework providing revision control services to connected CASE tools.

The model actions are:

- **NewElt**, creates a new model element in the model;
- **DelElt**, deletes an existing model element from the model;
- **NewRef**, creates a new model reference between two model elements;

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

– **DelRef,** deletes an existing model reference between two model elements;

– **AddVal**, adds a value to an attribute of a model element;

– **RmVal**, removes a value from an attribute of a model element.

The view actions are:

– **ShowElt,** includes a model element in a view;

– **HideElt,** hides a model element in a view;

– **ShowAttr**, display a model element attribute in a view;

– **HideAttrf**, hides a model element attribute in a view.

Model and view actions must be passed by the connected CASE tool to the galaxy instance framework. There is one artifact action class for each are operation of the three concrete artifact subclasses, *ProductUnit, ReuseUnit, MethodUnit* and their content, classes *ModelElt, ModelRef, Attribute and View*. Each operation of these concrete classes is realized by calling the *execute()* operation of the corresponding action class. For example, if *ru* is a *ReuseUnit* object and *me* a *ModelElt* object, then a call *ru.addMe(me)* is realized by the call *rua.execute()* where *rua* is a *RuAction* object which *ru* role is filled with *ru* and which *me* role is filled with *me.* This reification pattern allows an action to possess two dual aspects: a behavioral aspect encapsulated by the operation that it realizes and a data aspect that corresponds to the trace of calls to this operation.

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

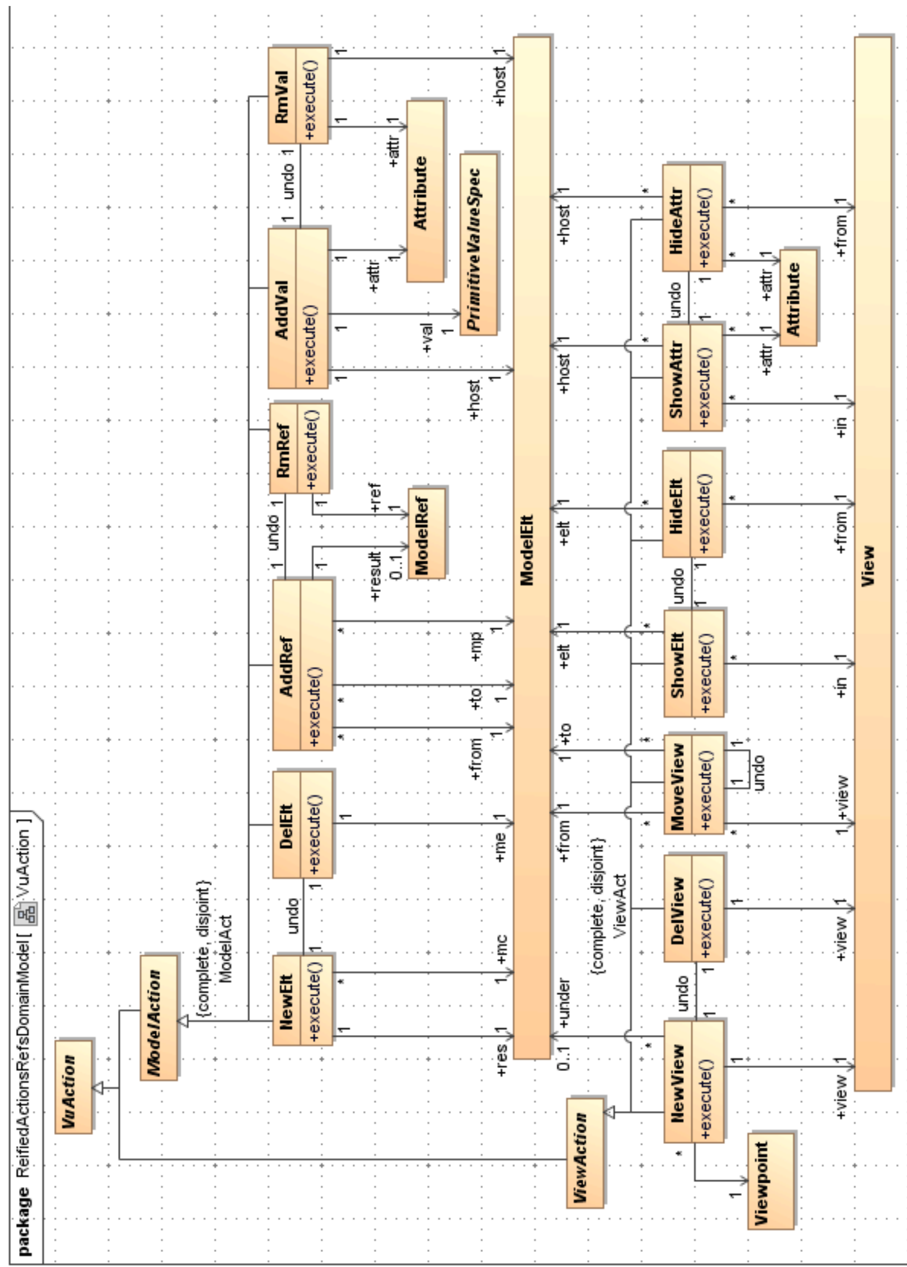

Figure 5: Minimal set of primitive model and view edition actions

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.

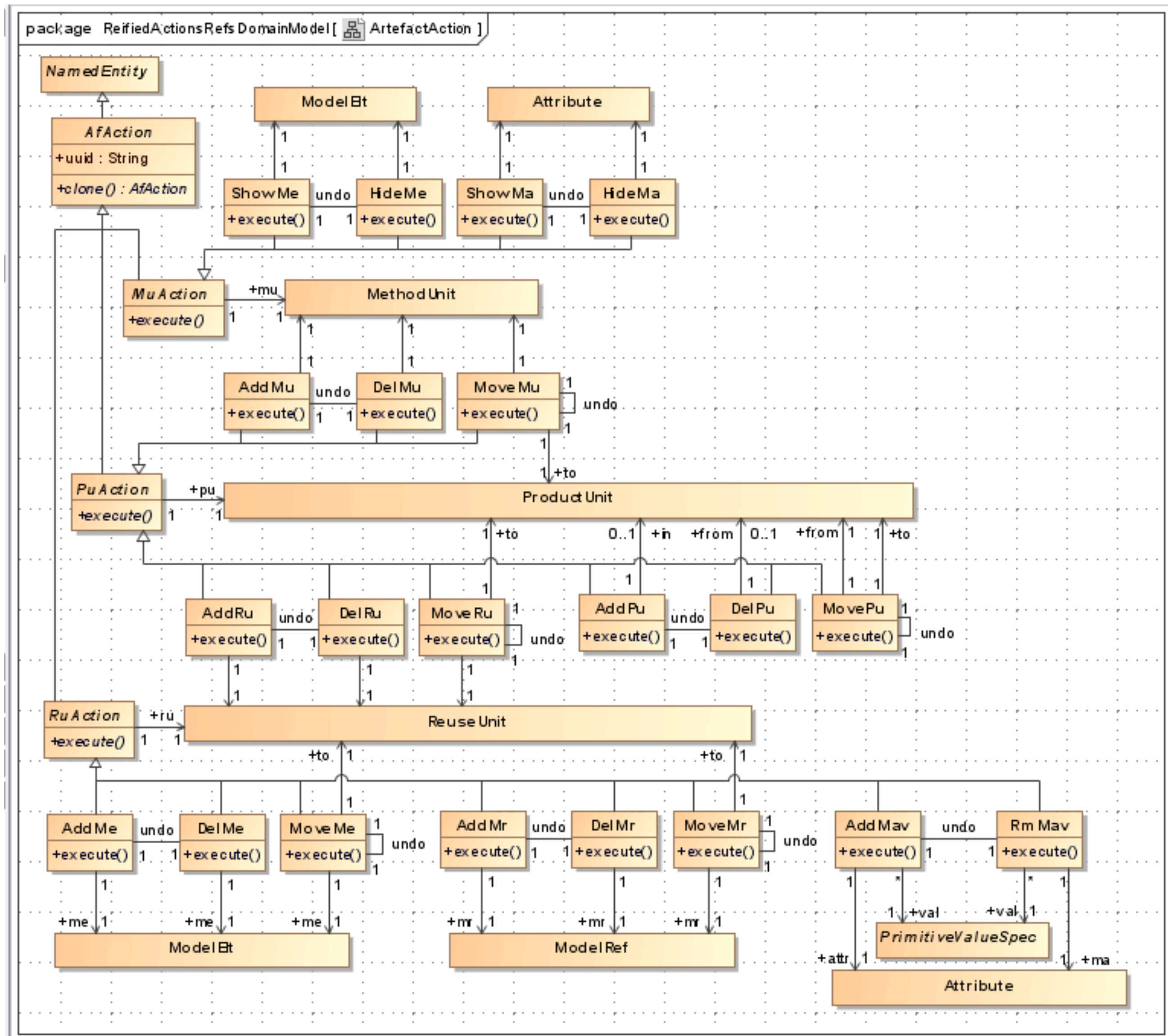

Figure 6: Minimal set of primitive artifact manipulation actions.

The artifact actions are:

- **AddPu**, adds a product unit toa container product unit; reifies the operation *add(afId: String)* of *CompositeArtifact* in Figure 1. when *afId* is the *uuid* of a *ProductUnit* object;
- **DelPu,** deletes a product unit from a container product unit; reifies the operation *del(afId: String)* of *CompositeArtifact* in Figure 1. when *afId* is the *uuid* of a *ProductUnit* object;

©Galaxy consortium, 2010. ALL RIGHTS RESERVED. CONFIDENTIAL AND PROPRIETARY DOCUMENT.