REuse and Migration of legacy applications to Interoperable Cloud Services
REMICS
Small or Medium-scale Focused Research Project (STREP)
Project No. 257793

Deliverable D3.1

REMICS KDM extension

Work Package 3

Leading partner: Netfective

Author(s): Franck Barbier, Gaëtan Deltombe, Alexis Henry

Dissemination level: PU

Delivery Date: August 31st, 2011

Final Version: 1.0

Versioning and contribution history

Version	Description	Contributors
0.1	Initial version	Franck Barbier
0.2	Reviewed version by Brice M. and Andrey S.	Franck Barbier
1.0	Final version	

Executive Summary

This document (D3.1) describes the *Extension of Knowledge Discovery Metamodel (EKDM)*, a tailored REMICS extension of the *Abstract Syntax Tree Metamodel (ASTM)* and *Knowledge Discovery Metamodel (KDM)* worldwide standards. Since *KDM* and *ASTM* are metamodels in the logic of the *Eclipse Modeling Framework*, *EKDM* is also a metamodel that merges these predefined metamodels along with appropriate innovative extensions developed in REMICS.

The emergence of model-driven modernization nowadays generates the first utilizations of both *ASTM* and *KDM*. In REMICS, the modernization of the DOME and DISYS case studies show lacks and an absence of maturity for both *ASTM* and *KDM*. That is the reason why *EKDM* is required to address model-driven modernization issues beyond REMICS.

This document describes the inner workings of *EKDM* while the D3.2 deliverable puts *EKDM* into practice.

Table of contents

1 Introduction

The ADM task force of the OMG has released several standards to carry out MDD reverse engineering activities and to support, more or less automatically, these activities in CASE tools.

In REMICS, the strategy is to seek a maximum of compliance to OMG standards and, if possible, to influence these existing standards through innovation and progresses made in REMICS/WP3 in particular. For that, REMICS/WP3 reuses the KDM standard whose main and recognized implementation is that from *kdmanalytics.com*.

More recently, the ADM task force has released ASTM in January 2011 to fill a gap in the daily use of KDM for legacy code manipulation especially. As shown later in this document (section 3.2), ASTM complements KDM regarding code issues. This document emphasizes code manipulation through KDM, ASTM and **the concomitant and coherent use of both through a REMICS homemade metamodel named EKDM (Extension of KDM)**. These results from the absence of a clear and formal interrelation specification between KDM and ASTM, **a lack of experience/lessons learned in the daily use of these cutting-edge technologies as well**.

As primitive material, KDM, ASTM and EKDM cannot be used in a friendly way outside the scope of a tool. In REMICS, BLU AGE® from Netfective (www.bluage.com), a proprietary tool, is based on this metamodel suite. D3.2 document provides information on what exactly is the REVERSE module of BLU AGE® along with a simple usage scenario of KDM, ASTM and EKDM in this commercial product. In D3.1 (this document), this is the specification of the first version of EKDM.

1.1 Terminology and Abbreviations

ADM	Architecture-Driven Modernization
ASTM	Abstract Syntax Tree Metamodel
CASE	Computer-Aided Software Engineering
CRUD	Create, Read, Update and Delete
EKDM	Extension of KDM
EMF	Eclipse Modeling Framework
KDM	Knowledge Discovery Metamodel
MDD	Model Driven Development
OMG	Object Management Group

1.2 Requirements traceability

D3.1 is part of WP3 whose original objectives are:

a) to define an integrated method for knowledge discovery to extract business value information from legacy including business models, components, implementation details and test specifications;

b) to specify the KDM extension to support the method;

c) to develop tools that supports the REMICS Recover process.

D3.1 fulfils the requirements as follows:

a)	EKDM focuses on code manipulation including data persistence-related code and UIs-related code. Next releases will focus on other software facets namely business rules.

b)	D3.1 gives the specification of the first version of EKDM, which will be enhanced in the remaining time of REMICS.
c)	BLU AGE® from Netfective has been updated to manage EKDM models; BLU AGE® generates UML models (with specific tags) as final outputs of the recover process. These models are used by Modelio™ and Metrino within, respectively, the WP4 and WP6 workpackages. Modelio™ and Metrino are been tested and adapted accordingly.

1.3 Work done in WP3/REMICS at M12

- REMICS provides an early implementation of ASTM that is available soon from the REMICS Web site (www.remics.eu) and right now from the REMICS eRoom as an EMF file: **ASTM.ecore**. The ability to properly deal with this metamodel requires the KDM implementation from *kdmanalytics.com*;
- **EKDM.core** which a coherent composition pattern and realization between KDM and ASTM. This metamodel file is also available from the REMICS eRoom and soon from the REMICS Web site;
- DOME case study also available from the REMICS eRoom provides instances of these three metamodels. However, any direct use of these metamodels in EMF is a tough task even impossible challenge due to the complex nature of KDM, ASTM and EKDM model instances in general. Instead, the final result of the reverse engineering of the DOME case study by means of BLU AGE® is supplied as UML user-friendly models in the REMICS eRoom but it is confidential to REMICS partners;
- DISYS case study will be available soon as a second illustration of EKDM.

2 REMICS tool chain organization

REMICS has established an integrated tool chain involving BLU AGE® from Netfective, Modelio™ from Softeam and Metrino from FOKUS. As shown in Figure 1, the reverse activity (left hand side) leads to UML Platform-Independent Models from which, Modelio™ and Metrino work.

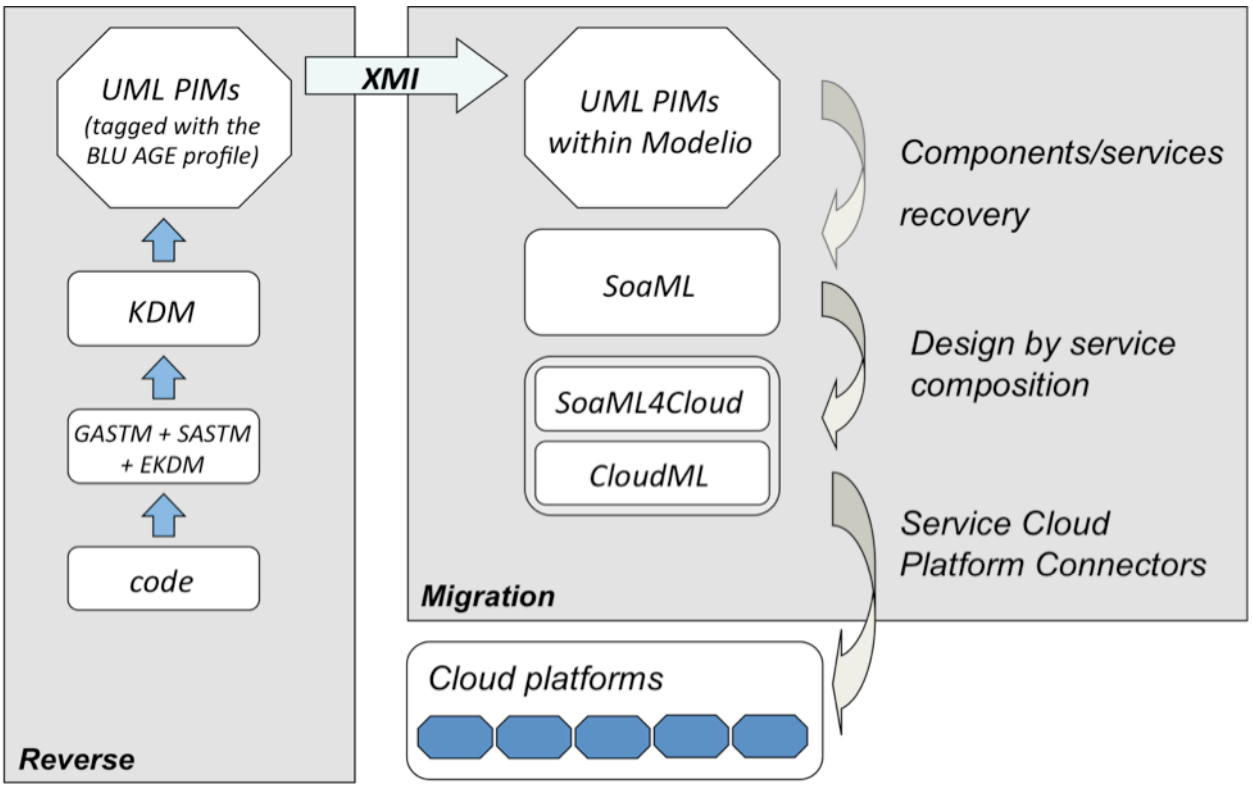

Figure 1 – REMICS tool chain organization

2.1 Status of tasks

- T3.1: results from this task are mainly part of this D3.1 document;
- T3.2: results are expected to be described in D3.5 with experimental implementations in D3.3 and D3.4;
- T3.3: Softeam recovers and re-factorizes component/service models based on UML models (Figure 1);
- T3.4: FOKUS recovers and processes test material based on UML models.

3 KDM, ASTM and EKDM

3.1 KDM

As its name suggests, ASTM is dedicated (through grammar manipulation) to the detailed management of programming language structure (including odd shapes of non-common programming languages like fourth-generation languages (4GLs) from the '80s or COBOL dialects). Instead, KDM is made up of various goal-oriented packages: "Micro KDM" but also Data, UI… (Figure 2).

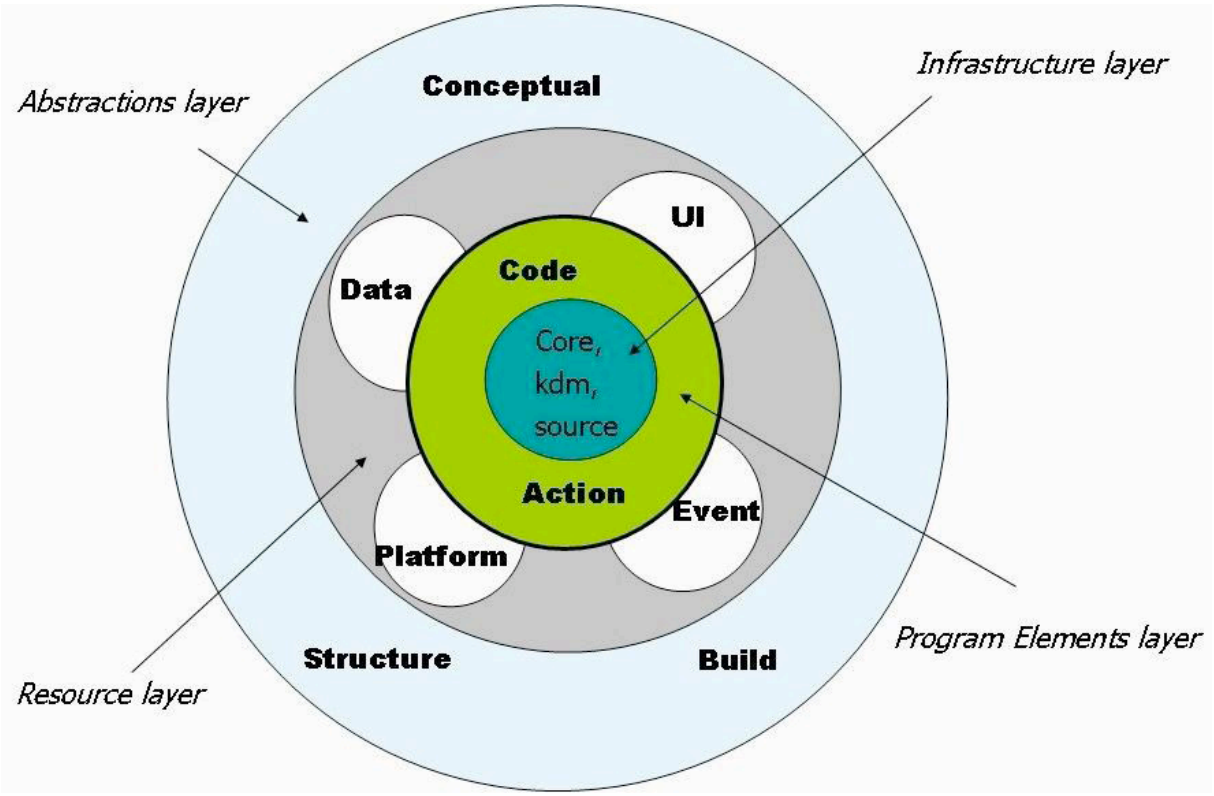

Figure 2 – KDM overview taken from the ADM/OMG documentation

In KDM, except "Micro KDM" (the two inner drunk areas), all surrounding layers are used for the modeling of features of legacy applications/information systems that have non-explicit links with the legacy code of interest.

While the *Core* and *kdm* packages provide infrastructure metaclasses and metarelationships for the all of the remaining packages, *Source* allows the recording of source material by means of the representation of traceability links between instances of KDM and code pieces in the legacy source code.

The *Code* and *Action* (second central drunk layer) together constitute what is called "Micro KDM" (or *Program Elements Layer*). Metaclasses and metarelationships in these two packages allow the representation of the legacy code in terms of elements, architecture between elements showing **data and control flows**, *i.e.*, algorithms. In essence, the detail level in KDM may change depending upon the chosen analysis degree (see also Section 3.3).

The *Code* package represents programming elements as viewed by programming languages (data types, procedures, classes, methods, variables, etc.) while the *Action* package describes the low-level behavior elements of applications, including control and data flows resulting from statement sequences, imbrications… In this scope, KDM is recognized as a way for representing the source code, only at the execution flow level.

In contrast, ASTM shows the code in the form of abstract syntax trees. In contrast to KDM, ASTM does **not** go beyond the source code. **The key (original) advantage of KDM is its ability to create a formal and rich link between code artifacts on one side and other legacy artifacts on another side towards the discovering and representation of business assets like business rules for instance**. For that, KDM *Resource layer* is composed of four packages: Data, UI, Event and Platform (we here concentrate on the *Data* and *UI* packages). *Data* is used to represent the organization of persistent data, especially to describe complex data repositories (*e.g.*, record files, relational databases…). *UI* is used to represent the structure of user interfaces and the dependency between them in terms of nesting, even interactions when the *Event* package is used.

3.2 ASTM

The concomitant use of KDM and ASTM imposes a clearer understanding of their current purposes and capabilities. In fact, as a complement to KDM, ASTM has been developed to support representation of source code at “procedure level”. In fact, ASTM is composed of both GASTM (*Generic Abstract Syntax Tree Metamodel*) and SASTM (*Specific Abstract Syntax Tree Metamodel*):

- GASTM allows the representation of the code without any language specificity. GASTM contains all of the common concepts of existing languages in the form of metaclasses. Parsing some code amounts to instantiating these metaclasses along with the creation of links in order to model semantic dependencies between code pieces. **The goal of GASTM is to provide an interface between KDM and SASTM; SASTM helps users to define domain-specific features of the code to be parsed**. The key achievement is to avoid an unintelligible separation (in terms of representations especially) between generic and specific characteristics of the code. Besides, the (annotated) distinction in models between generic versus specific parts, is highly valuable at processing time (see Section 3.3);
- SASTM is constructed through metaclasses/metarelationships on the top of GASTM. This task is assigned to ASTM practitioners. It first leads to constructing a metamodel from scratch that is compatible with the legacy language/technology to deal with. The main goal of SASTM is to represent code peculiarities. Next, parsing the code amounts to distinguishing between generic and specific aspects, and thus instantiating GASTM from SASTM models. The formal interrelation between the two metamodels ensures that models (their respective instances representing a given business case) are also consistently linked together based on (fully explicit) comprehensive links.

As a matter of fact, the difference between GASTM and SASTM resides in the abstraction level they provide when modeling the code of an application. SASTM supports a language-specific model of the initial application while GASTM supports a language-independent model of the initial application. Unlike GASTM, SASTM is in essence not standardized. It is a user-defined metamodel closely connected with a particular language (Java, COBOL and so on). Figure 3 illustrates some of the most familiar usages of ASTM.

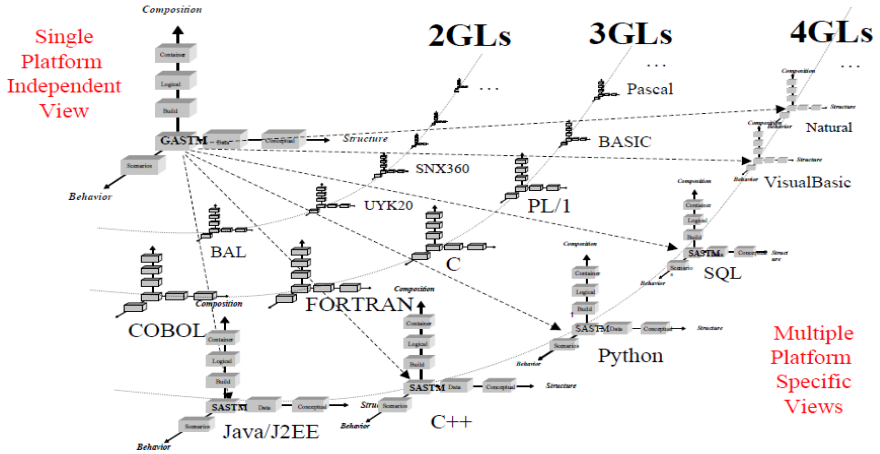

Figure 3 – GASTM versus SASTM from the ADM/OMG documentation

The complementarity of KDM and ASTM resides in the possible code level representations and the different transformations (in the spirit of MDD) that can be applied.

On one hand, ASTM enables to representing a given code source at procedure level, in the form of a syntax tree whose production involved a user-defined SASTM: code specificities are taken into account. On the other hand, KDM is used to represent the code at flow level (CRUD actions on data for instance). In fact, a flow level representation provides a direct support for flow analysis and refactoring strategies thereof. In addition, any reverse engineering process relying on KDM models is

reusable whatever the source technology is. So, ASTM deals with common parsing issues while KDM deals with another viewpoint, creating the link with other facets like user interfaces, components/services, architecture, business rules and so on.

3.3 KDM and ASTM in action

3.3.1 Code representation

Micro KDM enables to representing code with different granularity levels based on “macro actions” and “micro actions”. At parsing time, both concepts are reified as instances of the *ActionElement* KDM metaclass.

3.3.1.1 The notion of “macro action” and “micro action”

“Macro actions” describes several statements extracted from a legacy code of interest. By definition, “macro actions” deal with a higher abstraction degree when one wants to model a more or less sizeable execution section. Instead, “micro actions” show code behaviors with a greater zoom.

3.3.1.2 Open issues

Micro KDM has many drawbacks. Both ASTM and “micro actions” in Micro KDM have to stress code details from two distinct perspectives that are *a priori* a complementary perspective. Micro KDM falls short in some way. To demonstrate these drawbacks, let us consider the following example: a *if(1 == B)* execution condition (Figure 4).

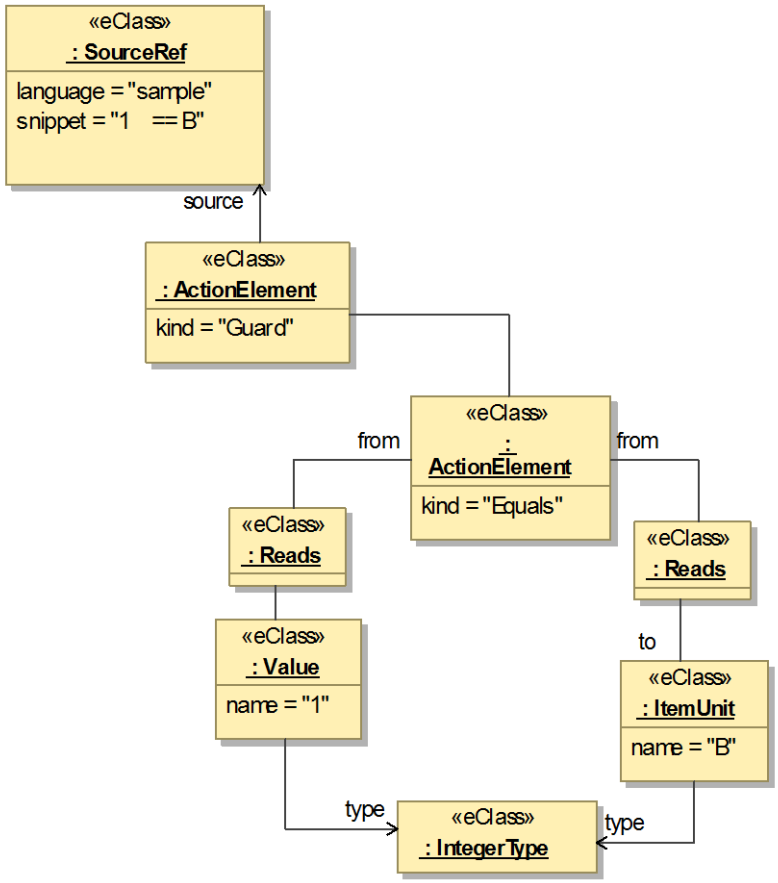

Figure 4 – KDM in action illustrating “micro actions”

The KDM instance diagram in Figure 4 represents the *if (1 == B)* condition. However, the representation of this condition via KDM provides **no additional information** compared to the ASTM representation (Figure 5). Moreover, the separation between the representation of the macro code and micro code is fairly vague. Besides, the semantics of operators in the KDM representation is supported by the *kind* attribute of *ActionElement* whereas, in ASTM, the semantics of operators is represented by instances of foreground metaclasses (*e.g.*, *BinaryExpression*).

So, the KDM representation has less quality than that of ASTM. Indeed, in KDM, the "1" element of the initial condition expression is represented by an instance of *Value*. The *name* attribute of *Value* symbolizes this "1" element. In ASTM, the "1" element" leads to *IntegerLiteral* instance with a *value* attribute set to "1".

Unfortunately, Micro KDM was designed to be able to be actually abstract from any implementation. Yet, it is not the case because micro actions from KDM do not provide a satisfactory abstraction level. There is no abstraction at all in KDM here and, above all, what is represented with KDM is already represented via an enhanced semantics in ASTM.

This example proves that KDM and ASTM have not been thought to properly work together.

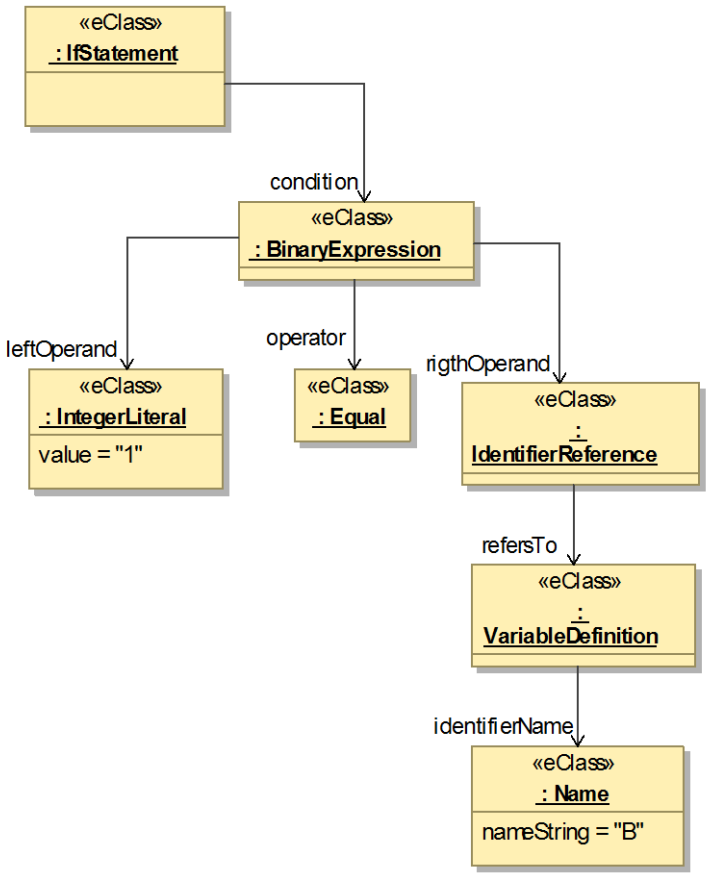

Figure 5 – ASTM in action

To sum up at this stage, in KDM, usages of micro actions are useless when ASTM is concomitantly put into practice; the cost of using KDM is not justified.

3.3.1.3 KDM as the pivot metamodel

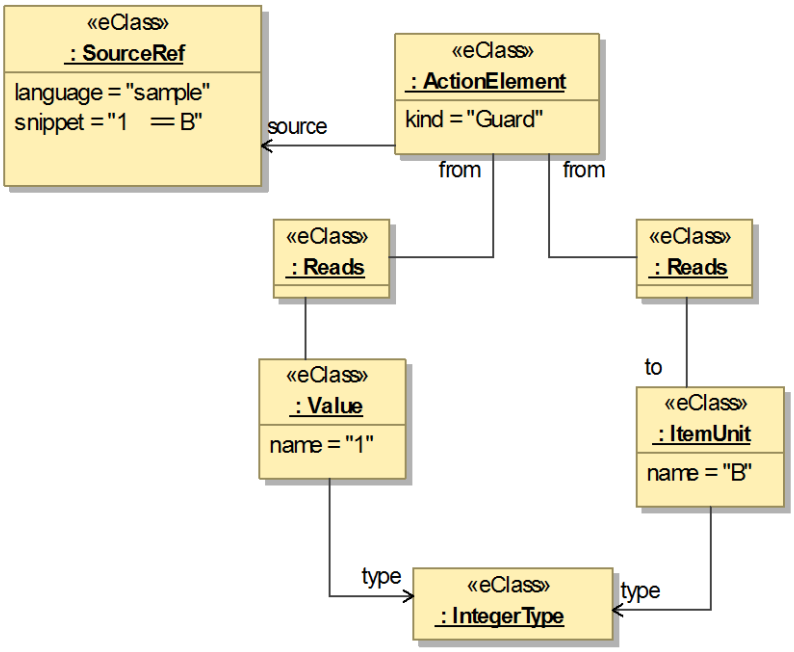

Figure 6 – Macro action illustrated in KDM

In contrast to micro actions, KDM macro actions are a valuable support when properly used. One is in particular able to design KDM models that bring a good abstraction degree (Figure 6). In Figure 6, the *Equal* metaclass instance has been removed from the model in Figure 4. This toy demonstration shows that KDM metaclasses like *ActionElement* are useful when following best practice principles. In this case, modeling operators like *Equal* has no value because ASTM does the same in Figure 5. In Figure 6, data and control flows embodied by "read" actions in this case, add useful information compared to the ASTM model in Figure 5.

Originally, KDM was intended to process code details but the arriving of ASTM invites to put aside micro actions. What is missing, is the coherent delimitations of both KDM and ASTM frontiers along with a formal interrelation (new metaclasses and/or metarelationships). **An adequate strategy is to accurately measure up to what extent ASTM will model code details and KDM will take over from ASTM** when coping with data, UIs and so on. For example, the legacy code of interest may contain mappings between declared data types and records in files/tables in databases. KDM enables to discovering and formalizing, when and how variables/objects of these types are created, read, updated and deleted. More precisely, even though KDM has some supports to model data and UIs for instance, **it has no support to model such mappings**.

At this stage, to summarize, ASTM allows the best (detailed) description of the code structure with respect to the programming language used even if the latter has odd features (*e.g.*, no context-free grammar). KDM adds value to the ASTM in terms of execution flows when one keeps a well-balanced abstraction level. KDM also permits the description of other software artifacts like persistence elements, I/O devices. However, no mapping with the execution flows is supported; no gateway to ASTM is also supported. This is the role of EKDM.

3.3.2 Mappings

3.3.2.1 Data

Dependencies between code pieces and data are concerned with data manipulation (CRUD operations), differentiating between temporary data and persistent (business) data. In short, mappings

play a great role when one wants to know and to capture links between execution flows and data management.

3.3.2.2 UIs

Dependencies between the legacy code of interest and the result of capturing UI elements is important. This aims at identifying data structures associated with UI element types and variables; it aims at isolating any code chunck performing interactions in the form of, if possible, events sent and received from consoles (green old-fashion screens in general) and devices as well.

To overcome varied problems linked to UIs, EKDM was specified and implemented for REMICS.

3.4 EKDM

3.4.1 Motivations behind EKDM

Observations above stress the necessity to take advantage of the two worlds: KDM and ASTM. This means keeping both the support for data modeling, UI modeling, platform modeling (*e.g.*, legacy OS calls, legacy platform service usages)… provided by KDM and the level of precision for old-fashion language modeling allowed by ASTM. Therefore, we suggest to merging GASTM (the predefined component of ASTM) and KDM together in a whole that we dubbed EKDM. From this viewpoint, EKDM sets the balance between precision, abstraction and semantics. To put the KDM-ASTM merging into practice, a customized GASTM implementation version is required. As written above, this implementation appears in the **ASTM.ecore** file recorded in REMICS eRoom.

Having EKDM as a set of additional metaclasses and metarelationships is not enough, one also needs a set of predefined model transformation along with a well-codified transformation process to move from one model to another. This process and the set of transformations is the heart of the BLU AGE® REVERSE engine. Since all of this material is a trade mark, it is not explained here in further details (see D3.2 for a brief “Getting started with EKDM”). Despite this confidentiality, one must notice that transformations from ASTM to KDM sometimes require decomposition of some elements into more atomic elements. Doing this way, there is often a risk that the precise semantics held by an element is lost since rephrased into a more generic format. In all cases, such decomposition is not systematic. Sometimes, the semantics tied to a specific technology cannot always be kept for its subsequent processing. As a result, a good balance between abstract (or KDM-based) and generic (GASTM-based) representations is sought. Beyond, all transformations cannot be automated calling for human intervention when necessary.

3.4.2 EKDM status and overview

At this stage, EKDM only relies on *Micro KDM*. Forthcoming REMICS deliverables (D3.3 and D3.4 which are products and D3.5 “REMICS Recover Principles and Methods” which is a report) will show enhanced versions of EKDM to create gateways to, respectively, the KDM *Resource layer* and the search for an alignment with the *Semantics of Business Vocabulary and Business Rules* (www.omg.org/spec/SBVR/1.0) standard from the OMG/ADM.

EKDM offers a coherent articulation pattern (new metaclasses and new metarelationships based on a canonical format) between KDM and ASTM while the interrelation between these two standards is completely fuzzy, even undefined, in the OMG/ADM specification. EKDM benefits from early validation related to industrial representative applications of Netfective clients in the USA, DOME and DISYS case studies as well. Indeed, DOME and DISYS case studies are recovered by means of these metamodels (KDM, ASTM and EKDM) along with expectations about (forthcoming) innovative extensions imposed by these two case studies, even case studies belonging to the REMICS extension project.

To be compatible with EKDM, tools must offer all of the metamodel elements appearing in Figure 7 and in Section 3.4.3. This covers the following capabilities:

- Tools must have the ability to generate a XMI document as an instance of the XMI EKDM source text provided in **EKDM.ecore**;

- Tools must have the ability to import a XMI document as an instance of the XMI EKDM source text provided in **EKDM.ecore**.

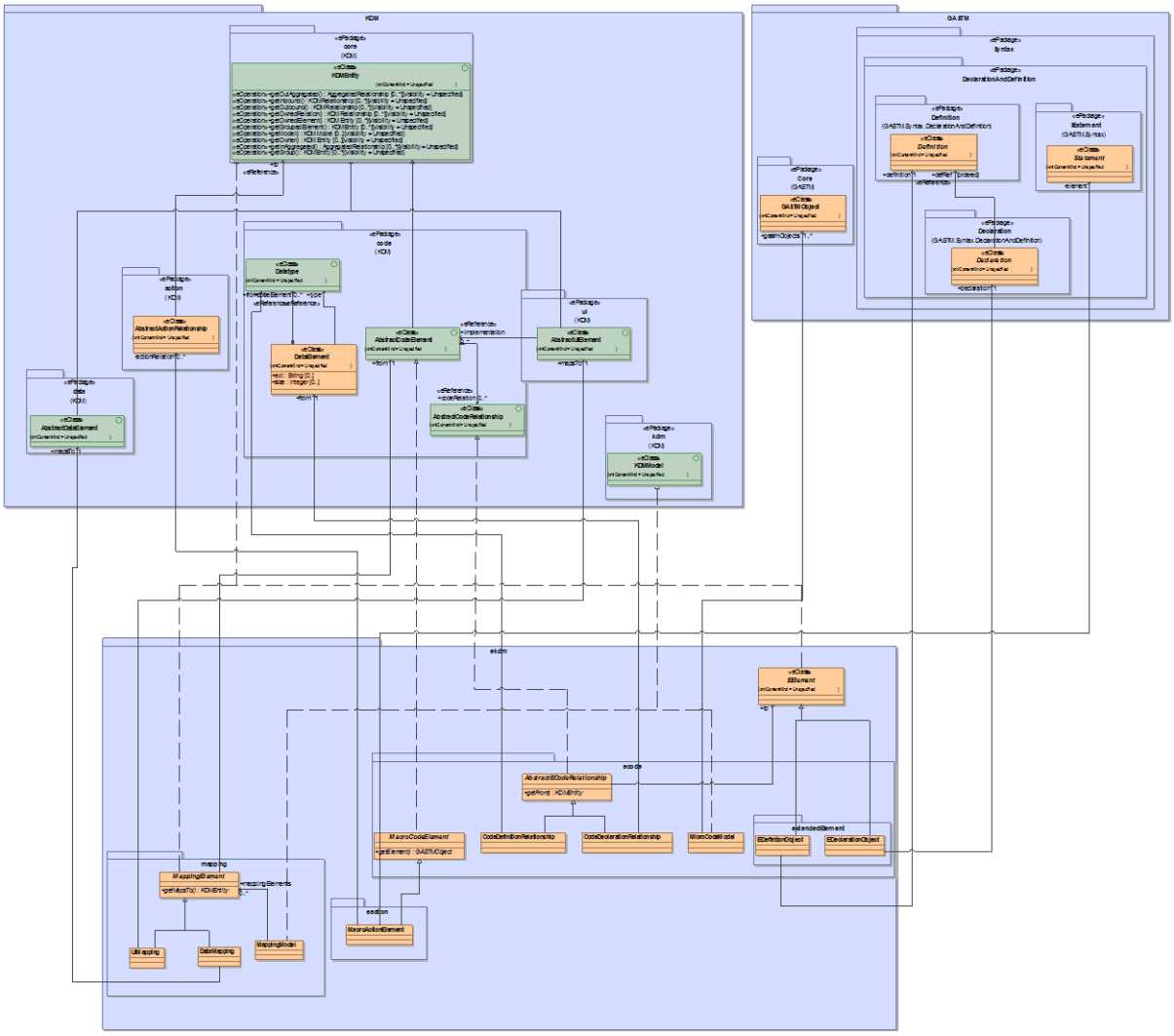

Figure 7 – EKDM overview (details are in Section 3.4.3)

3.4.3 *ekdm* package description

3.4.3.1 EElement

3.4.3.1.1 Description

The *EElement* abstract metaclass represents the interface that creates a mapping from an object in the KDM *Code* package and an external object. Such an object belongs to “external” metamodels (typically, SASTM instances) when one needs homemade metamodels to cope with eclectic facets of the legacy code of interest.

3.4.3.1.2 Specification

The *EElement* metaclass inherits from the *KDMEntity* metaclass.

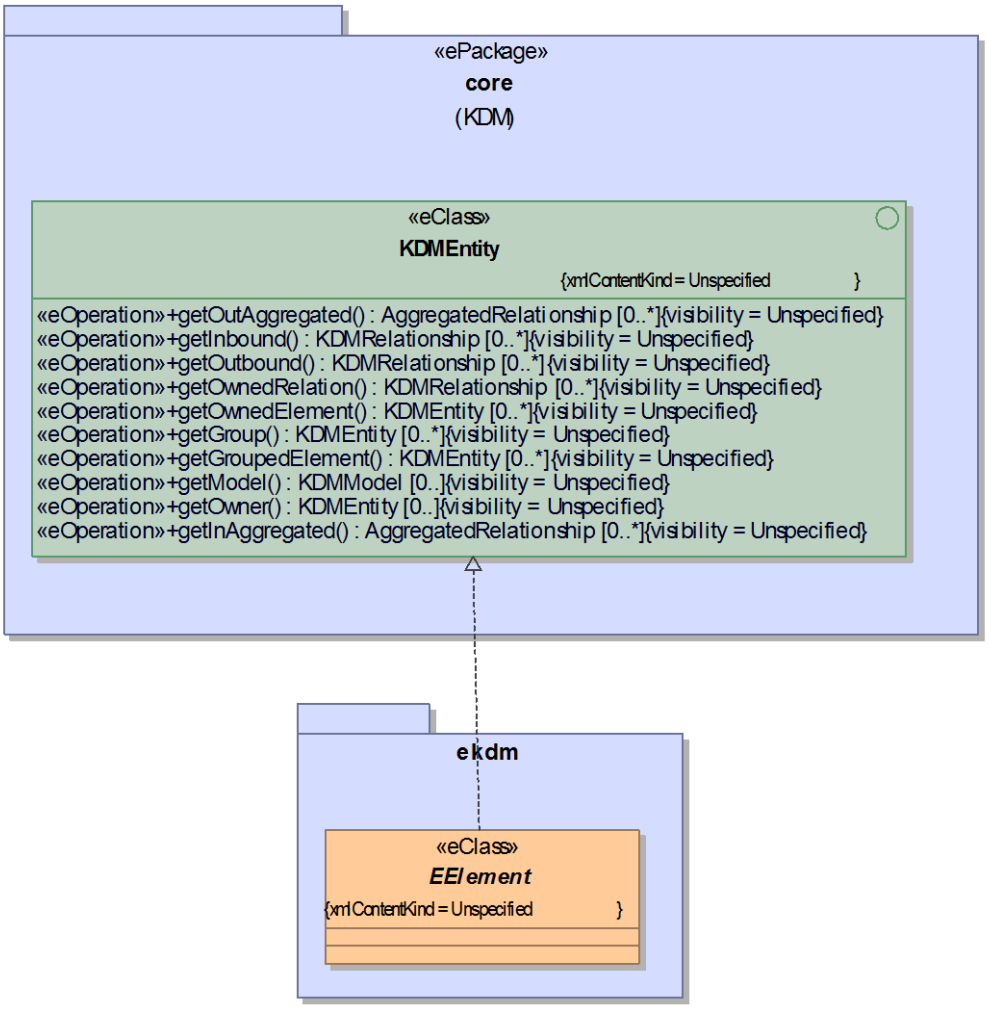

Figure 8 – *EElement* specification

3.4.4 *ecode* package description

3.4.4.1 MicroCodeModel

3.4.4.1.1 Description

The *MicroCodeModel* metaclass aims at encapsulating the representation of ASTM within KDM.

3.4.4.1.2 Specification

This metaclass implements the *KDMModel* interface belonging to the *kdm* package. It possesses a collection of instances of the *GASTMObject* metaclass.

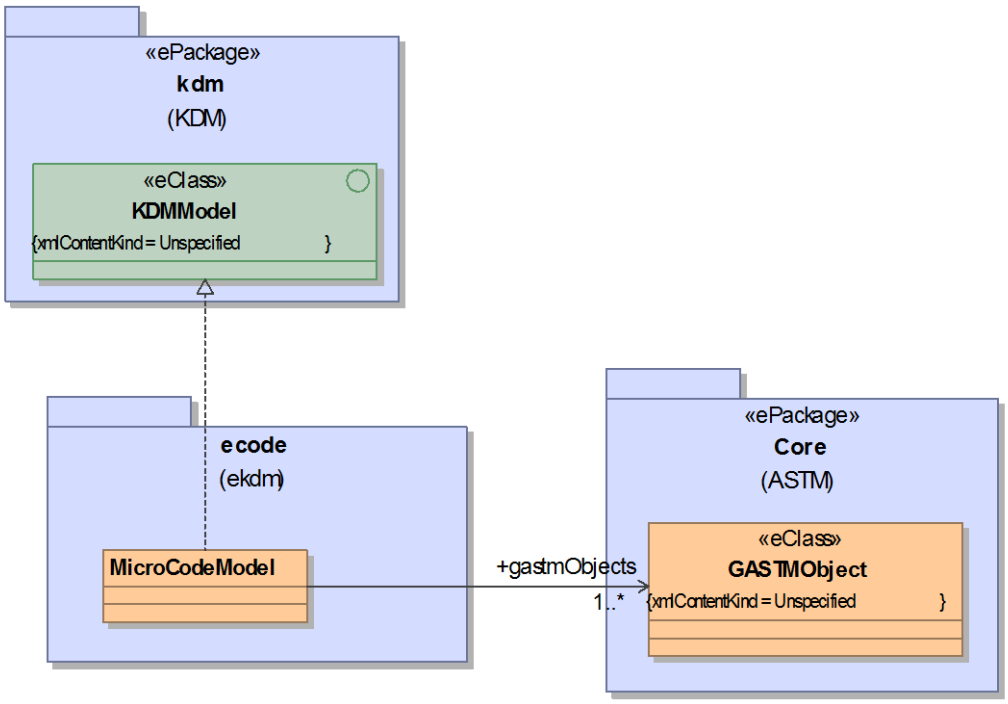

Figure 9 – *MicroCodeModel* specification

The code micro-level representation is assigned to ASTM and is encapsulated in KDM. The global linking between the “micro” and “macro” layers is materialized by the *MacroCodeElement* abstract metaclass.

3.4.4.2 MacroCodeElement

3.4.4.2.1 Description

The role of this abstract *MacroCodeElement* metaclass is the provision of a uniform interface that guarantees the formal linking of the “micro” and “macro” representations of the legacy code of interest.

3.4.4.2.2 Specification

The *MacroCodeElement* metaclass is an abstract class. It offers a *getElement()* abstract method which returns a *GASTMObject* object. This method embodies the link between the “micro” and “macro” representations of the legacy code of interest.

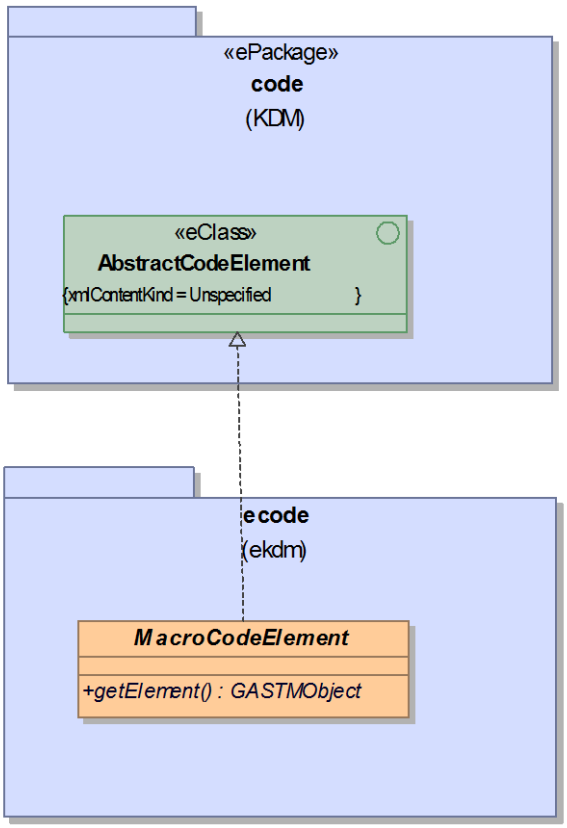

Figure 10 – *MacroCodeElement* specification

3.4.4.3 AbstractECodeRelationship

3.4.4.3.1 Definition

The *AbstractECodeRelationship* abstract metaclass defines a uniform interface allowing semantic relationships between the representation of macro code in KDM and that of “external” metaclasses from a metamodel different from KDM (typically, SASTM).

3.4.4.3.2 Specification

The *AbstractECodeRelationship* abstract metaclass inherits from the *AbstractCodeRelationship* metaclass. It possesses a reference to an *EElement* object. It also supports a *getFrom()* abstract method which returns a *KDMEntity* object.

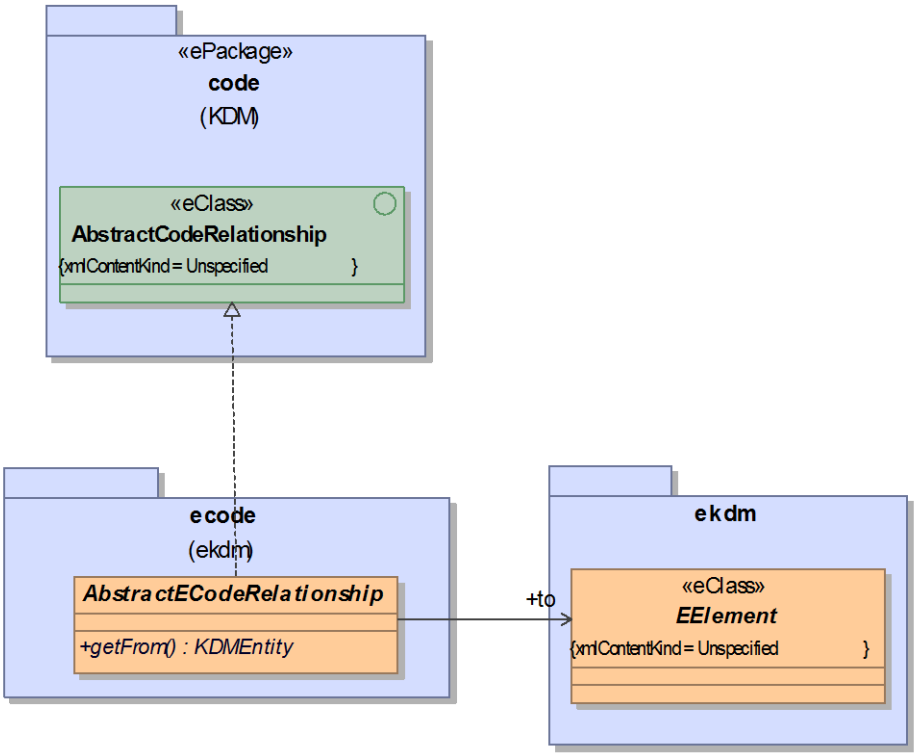

Figure 11 – *AbstractECodeRelationship* specification

3.4.4.4 CodeDefinitionRelationship

3.4.4.4.1 Definition

The *CodeDefinitionRelationship* metaclass allows the definition of a relationship between the “micro” representation of a *Definition* object and its “macro” representation in KDM.

3.4.4.4.2 Specification

The *CodeDefinitionRelationship* metaclass inherits from the *AbstractECodeRelationship* metaclass.

This metaclass allows the linking of an *EElement* object and a KDM *Datatype* object

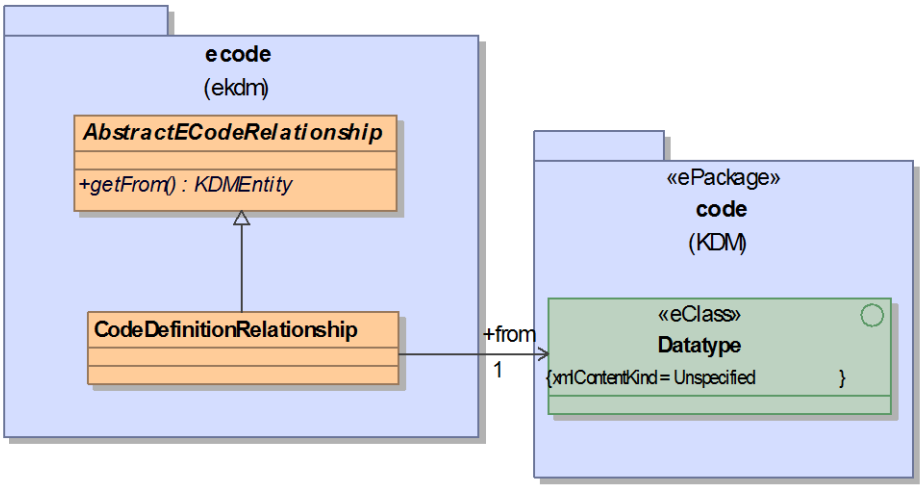

Figure 12 – *CodeDefinitionRelationship* specification

3.4.4.5 CodeDeclarationRelationship

3.4.4.5.1 Definition

The *CodeDeclarationRelationship* metaclass allows the definition of a relationship between the "micro" representation of a data type declaration in ASTM and its "macro" representation in KDM.

3.4.4.5.2 Specification

The *CodeDeclarationRelationship* metaclass inherits from the *AbstractECodeRelationship* metaclass.

This metaclass allows the linking of an *EElement* object and a KDM *DataElement* object.

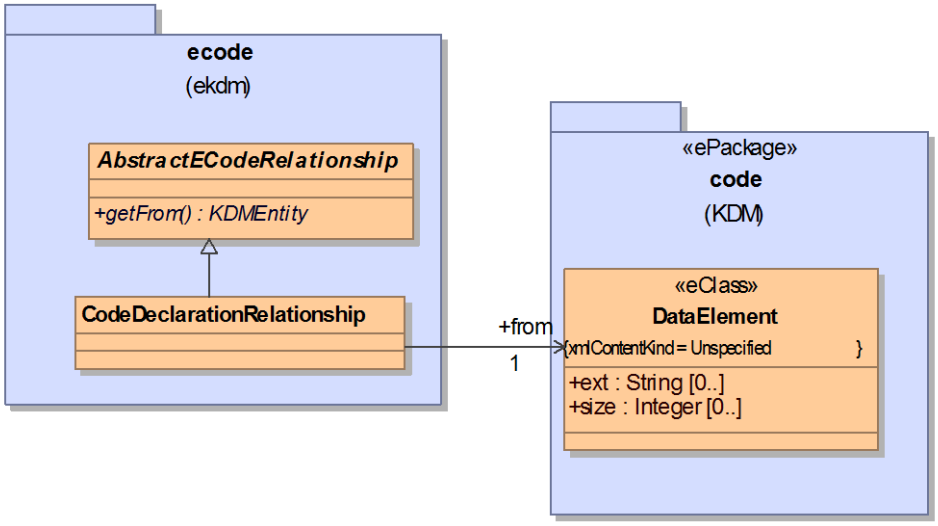

Figure 13 – *CodeDeclarationRelationship* specification

3.4.5 *extendedElement* package description

The *extendedElement* package contains the definition of "extended elements". These elements are not KDM elements. Extended elements allow the adaptation of elements from other metamodels (typically, SASTM) to KDM.

3.4.5.1 E(xtended)DeclarationObject

3.4.5.1.1 Definition

The *EDeclarationObject* metaclass allows the adaptation of ASTM objects whose type is *Declaration* to KDM. One may typically here observe that EKDM offers a proper gateway between ASTM and KDM beyond the fact that EKDM supports missing facilities in both KDM and ASTM.

3.4.5.1.2 Specification

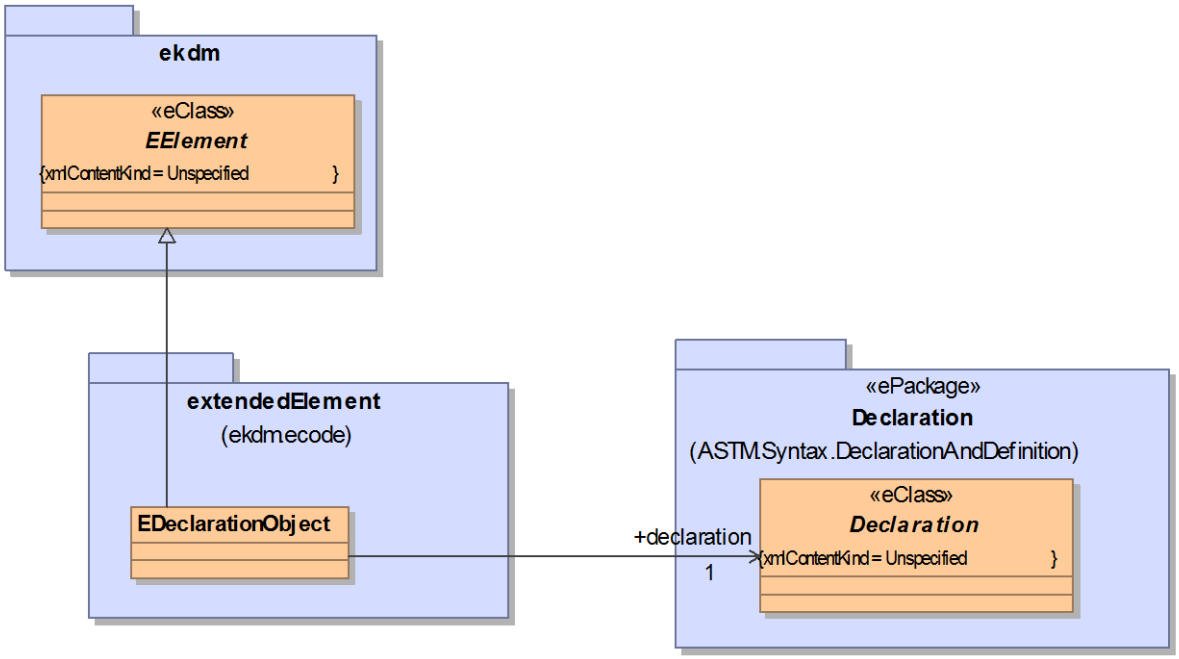

Figure 14 – *EDeclarationObject* specification

3.4.5.2 E(xtended)DefinitionObject

3.4.5.2.1 Definition

The *EDefinitionObject* metaclass allows the adaptation of ASTM objects whose type is *Definition* to KDM.

3.4.5.2.2 Specification

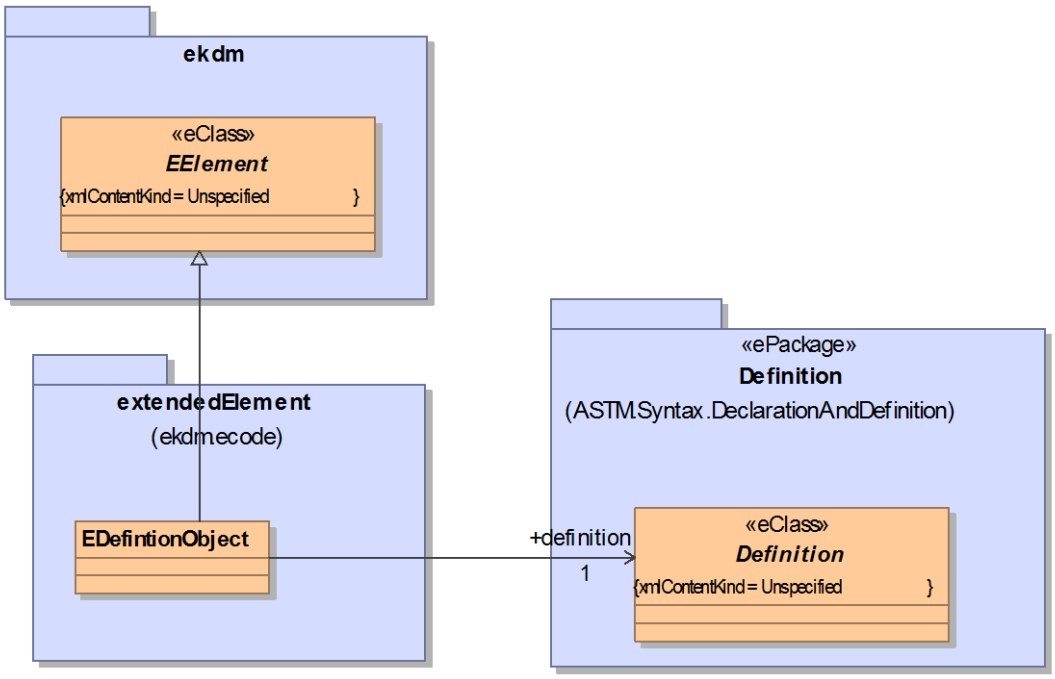

Figure 15 – *EDefinitionObject* specification

3.4.6 *mapping* package description

The *mapping* package in EKDM allows a mapping creation between data structure definition and one or more KDM elements belonging to its *Data* or *UI* predefined domain.

3.4.6.1 MappingModel

3.4.6.1.1 Definition

The *MappingModel* metaclass contains the mappings' definition.

3.4.6.1.2 Specification

This metaclass implements the *KDMModel* interface. It has elements that are instances of the *MappingElement* metaclass.

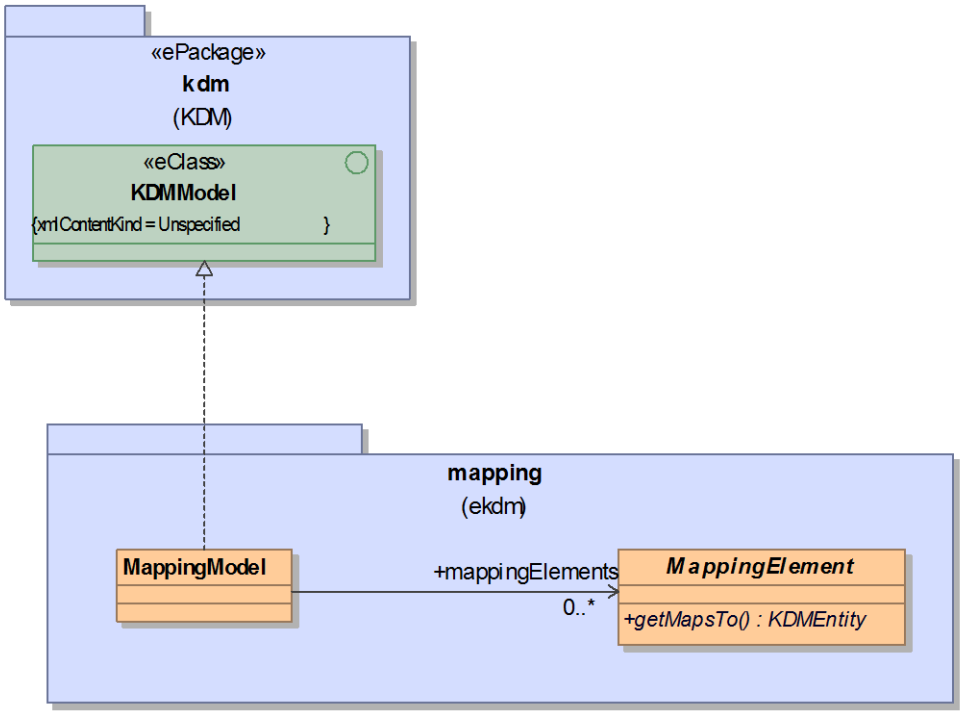

Figure 16 – *MappingModel* specification

3.4.6.2 MappingElement

3.4.6.2.1 Definition

The *MappingElement* abstract metaclass represents a common interface enabling the definition of a mapping between an *AbstractCodeElement* object (*i.e.*, any object of the KDM *Code* predefined domain) and a KDM entity in general.

3.4.6.2.2 Specification

This metaclass implements the *KDMEntity* interface. It possesses an association whose type is *AbstractCodeElement*. It also offers a *getMapsTo()* abstract method to return this object.

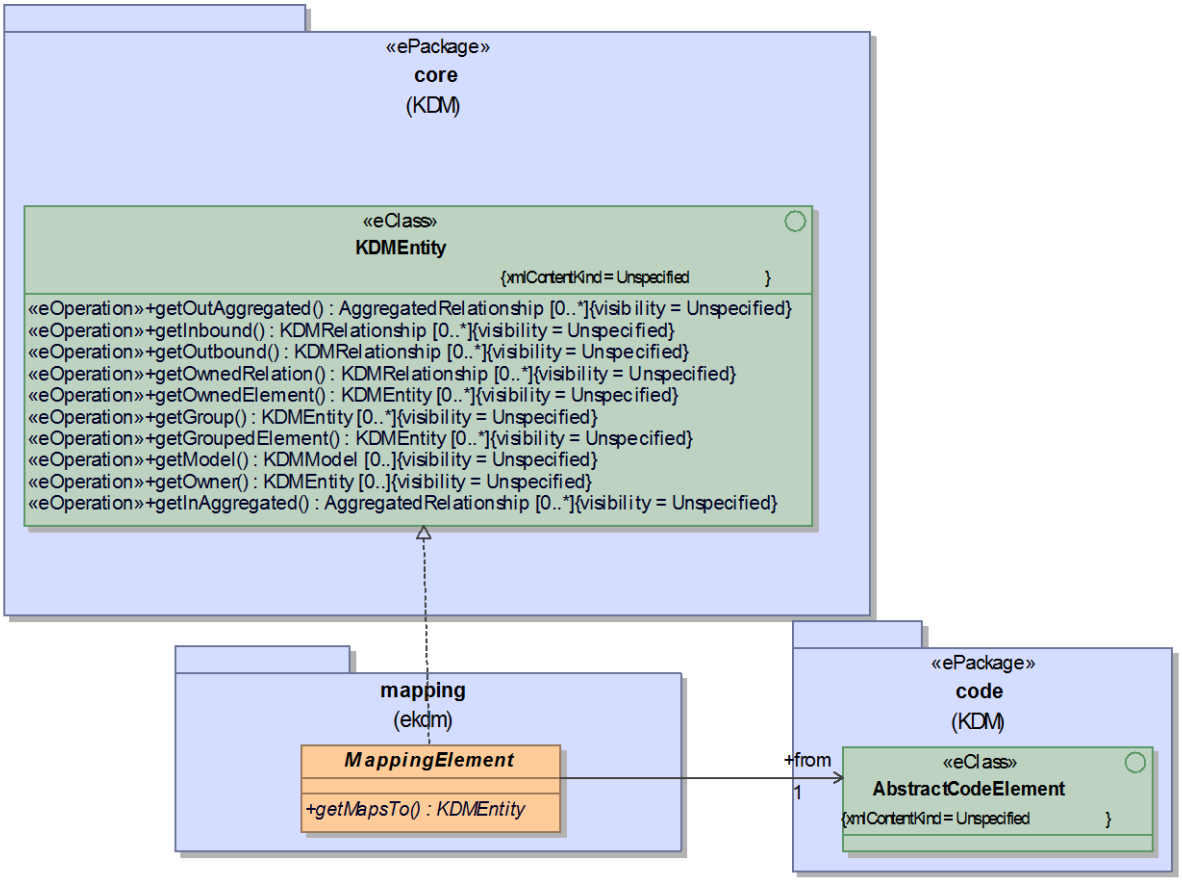

Figure 17 – *MappingElement* specification

3.4.6.3 UIMapping

3.4.6.3.1 Definition

The *UIMapping* metaclass supports mapping between any definition in the KDM *Code* domain and any object from the *UI* domain in KDM.

3.4.6.3.2 Specification

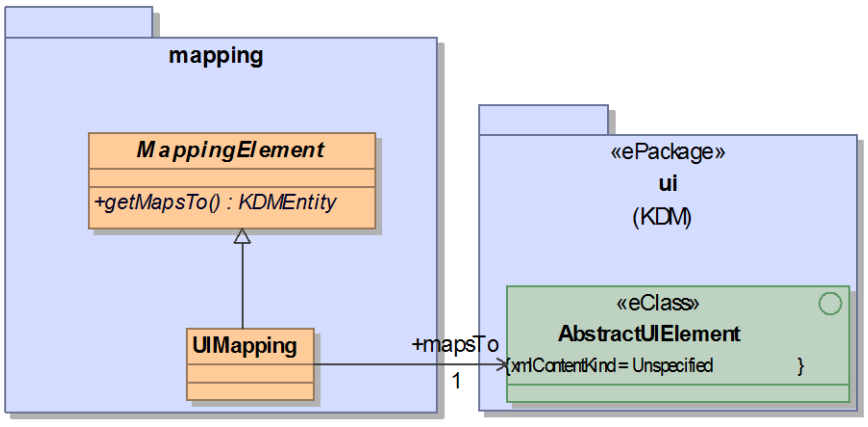

Figure 18 – *UIMapping* specification

3.4.6.4 DataMapping

3.4.6.4.1 Definition

The *DataMapping* metaclass supports a mapping between any definition in the KDM *Code* domain and any object from the *Data* domain in KDM.

3.4.6.4.2 Specification

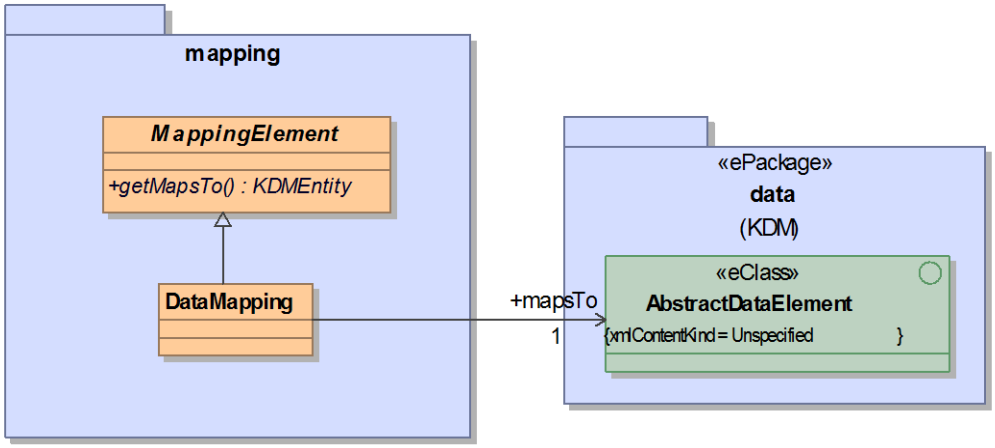

Figure 19 – *DataMapping* specification

3.4.7 *eaction* package description

The *eaction* package is an extension of the *Action* package in KDM. This package contains an *ActionElement* metaclass that both deal with “macro” actions in KDM. Note that this package currently contains only one metaclass.

3.4.7.1 MacroActionElement

3.4.7.1.1 Definition

The *MacroActionElement* metaclass enables to representing one or more statements from the ASTM initial representation (macro action).

3.4.7.1.2 Specification

This metaclass inherits from the *MacroCodeElement* metaclass. It possesses an attribute whose type is *Statement* (this is an encapsulation of the ASTM *Statement* metaclass). It also possesses a list of *AbstractActionRelationship* objects. These are used to describe in a more abstract way, the behaviour of any KDM *Action* object. Their role is the description of semantic relationships.

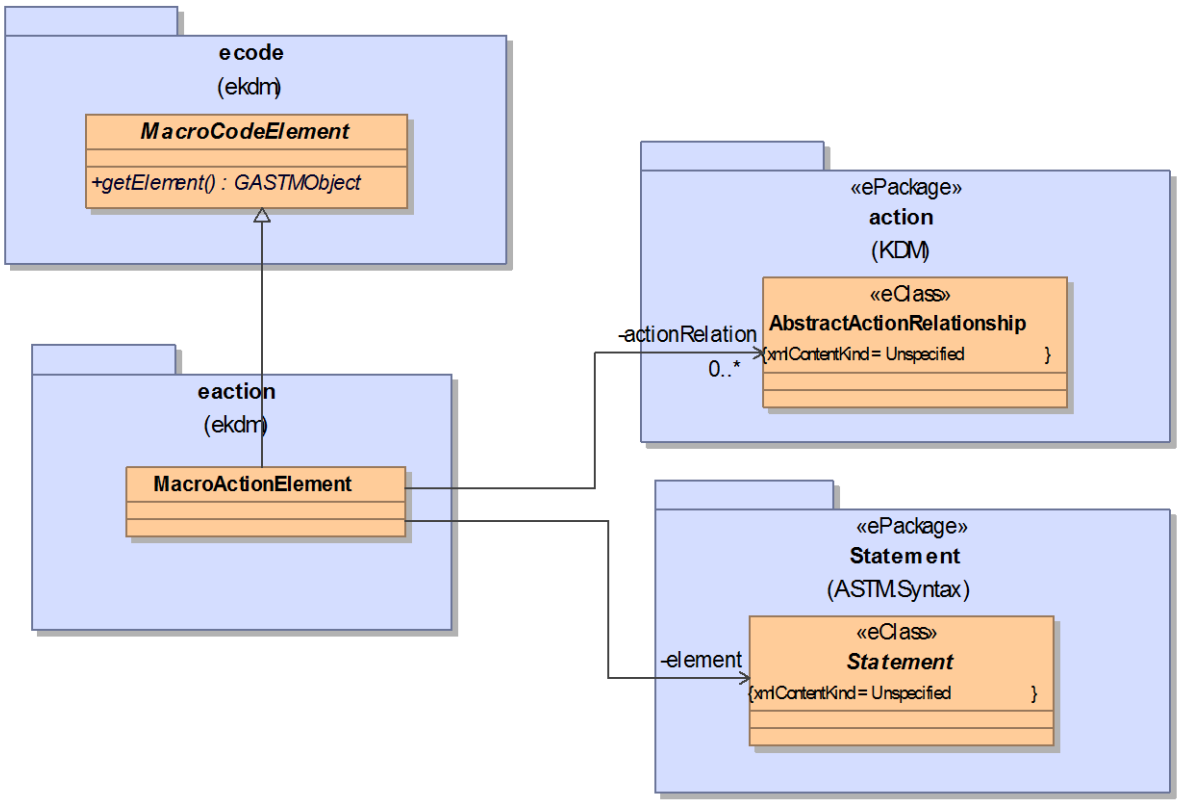

Figure 20 – *MacroActionElement* specification

4 Conclusions and ongoing work

EKDM offers a coherent articulation pattern (new metaclasses and new metarelationships) between KDM and ASTM while the interrelation between these two standards is unsound in the OMG/ADM specification. Moreover, EKDM benefits from an early validation related to *industrial representative* cases including DOME and DISYS case studies. These REMICS demonstrators are recovered by means of KDM, ASTM and EKDM metamodels along with expectations about (forthcoming) innovative extensions imposed by these two case studies. One may also notice that that two tailored SASTM metamodels have been developed to, respectively, process DOME PL/SQL dialect and DISYS Microfocus COBOL dialect.

So, ASTM and KDM complement each other in modeling software systems' syntax and semantics thanks to EKDM. In this D3.1 deliverable, we propose to bridge the gap between the two by expressing ASTM as a well-integrated extension of KDM in order to scrupulously follow the OMG/ADM spirit; hence, ensuring interoperability of OMG/ADM reverse engineering activities. Gluing ASTM and KDM aims at overcoming the main flaw of KDM, which is an inadequate representation of code for high-level analysis and transformations, and to provide a good abstraction level for code multiple-view representation. EKDM has been implemented within BLU AGE® and has been proven to better represent code while keeping a good architecture representation, rather than using KDM and ASTM separately. In this context, a main issue lies in tool interoperability since EKDM is not a standard. However, this can be solved at this time by "simple" transformations in order to first obtain KDM models and finally UML models (see Figure 1).

Globally, this proposal differs from the everything-is-transformation precept promoted by OMG. Instead, we first single out inheritance-based mappings and next complete the resulting metamodels by means of translation-based mappings. These are developed case by case because new requirements may call for adaptation of the transformation programs.

A debate is still raging in MDD communities about DSLs, metamodels versus profiles, to work out when one should create a new metamodel and when one should instead create an extension. Considering that ASTM is a DSL for representing source code at procedure level, the question rose again. For the time being, OMG/ADM decided that it was preferable to create a new metamodel completely separated from KDM. Yet, the extensibility mechanism of KDM allows the definition of a profile, that is, a set of stereotypes and tagged values associated with existing elements. By considering ASTM as a direct specialization of KDM, our proposal falls in this field of extension.

For the next REMICS two years, challenges are:

- How much of the recovery process might be automated? For instance, how many transformations might be written once and for all with an actual independence of the legacy application/information system and the legacy technology (programming language especially) in general?
- How much the recovery process will be able to make requirements, business intelligence… emergent in seamless and friendly way? We mean how to extract and properly reformat conceptual legacy artifacts to ease the generation of modernized applications/information systems managed in the form of UML PIMs?