Computer Standards & Interfaces 33 (2011) 519–532

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems

Ricardo Pérez-Castillo *, Ignacio García-Rodríguez de Guzmán, Mario Piattini

Alarcos Research Group, University of Castilla-La Mancha, P° de la Universidad, 4 13071, Ciudad Real, Spain

ARTICLE INFO

Article history:
Received 27 April 2010
Received in revised form 15 February 2011
Accepted 21 February 2011
Available online 27 February 2011

Keywords:
KDM
ISO19506
Knowledge management
Software modernization
Legacy system

ABSTRACT

Legacy systems age over time as a consequence of uncontrolled maintenance, thus they must be evolved while its valuable embedded knowledge is preserved. Software modernization, and particularly Architecture-Driven Modernization, has become the best solution in the legacy systems' evolution. ADM defines the Knowledge Discovery Metamodel specification, now being adopted as ISO/IEC 19506 by the International Standards Organization. The KDM metamodel allows to represent all the software artifacts recovered during reverse engineering techniques at different abstraction levels. This paper presents how to use KDM to modernize legacy systems, making them more agile, preserving the embedded business knowledge and reducing maintenance costs.

1. Introduction

As the history of software engineering reveals, information systems are not static entities, but changeable over time. Information systems degrade and age, and they become *legacy* information systems because the code of these systems was written long ago and now may be technologically obsolete [49]. This problem is known as the software erosion phenomenon [52]. Most often, it is due to maintenance activities carried out over time, since the successive maintenance changes in an information system minimize its quality. In this case, a new and improved system must replace the previous one. However, replacing these systems completely from scratch is very expensive, and also slows down the achievement of ROI (Return of Investment) [48]. Additionally, the software embeds a significant amount of business knowledge over time that would be lost if entirely replaced [44].

In tackling the software erosion phenomenon, evolutionary maintenance is the best solution for obtaining improved systems without discarding the existing systems, which minimizes software erosion effects. Evolutionary maintenance addresses adaptive and perfective maintenance changes [17], and makes it possible to manage controllable costs and preserve the valuable business knowledge embedded in the legacy system.

Reengineering has been the main mechanism for addressing the evolutionary maintenance of legacy systems for a considerable time [8]. Reengineering preserves the legacy knowledge of the systems and makes it possible to maintain software easily at a low cost [7]. This reengineering process consists of the examination and alteration of a legacy system to reconstitute it in a new form and the subsequent implementation of the new form [10].

Nevertheless, a 2005 study [48] states that over 50% of all reengineering projects currently fail, due to two main weaknesses: formalization and automation problems. On the one hand, reengineering processes lack formalization and standardization [21], since these processes are carried out in an *ad hoc* manner. Thus, different reengineering tools that address specific tasks in the reengineering process cannot be integrated or reused in different reengineering projects. And on the other hand, the reengineering of large complex legacy information systems is very difficult to automate [9]; since the reengineering processes cannot be repeatable, as a consequence of the formalization problem. Therefore the cost of maintenance based on reengineering grows significantly.

At this time, software modernization is a new specific kind of evolutionary maintenance paradigm to solve reengineering problems. ADM (Architecture-Driven Modernization) as defined by OMG (Object Management Group) [36], advocates carrying out the reengineering process but considering model-driven development principles. That is, ADM deals with all the involved software artifacts as models in a homogenous manner, and it facilitates the formalization of deterministic transformations between those models [33]. The model transformations can be formalized by means of QVT (Query/Views/Transformations), the model transformation language proposed by OMG [39]. The automation of the model transformations together with the model-driven development principles makes it

* Corresponding author. Tel.: +34 926295300; fax: +34 926295354.
E-mail addresses: ricardo.pdelcastillo@uclm.es (R. Pérez-Castillo), ignacio.grodriguez@uclm.es (I.G.-R. de Guzmán), mario.piattini@uclm.es (M. Piattini).

0920-5489/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.csi.2011.02.007

possible to reuse the models involved in ADM-based processes. As a consequence, the automation problem can also be solved due to the automated transformations together with the reuse of the models.

As part of the effort undertaken by the ADM Task Force of OMG, KDM (Knowledge Discovery Metamodel) is the first standard within a broad set of proposed standards [43], although many of these standards are still in the approval or development stage. KDM forms the core of that set of standards for two main reasons: (i) KDM addresses the main challenges that appear in the modernization of legacy information systems; and (ii) it is the cornerstone of the set of proposed standards, since the other standards are defined around KDM. Despite the fact that KDM had been defined by the OMG, it is being adopted as the international standard ISO/IEC 19506 ADM KDM [19] through the so-called "fast-track" liaison relationship between OMG and ISO.

KDM is a specification of agreed upon facts and how these facts are represented in XML by using OMG standards such as Model Object Facility (MOF) and XML Metadata Interchange (XMI). The availability of this information represented according to the KDM specification makes it possible (i) to store the facts about information systems in a compliant structure; (ii) to analyze and reason about KDM facts; (iii) to exchange KDM facts as XML documents; and (iv) to build tools so that parsing source code in a given programming language to generate language-independent and vendor-neutral facts about information systems.

From the modernization perspective, KDM allows addressing all the stages of modernization processes based on ADM: reverse engineering, restructuring and forward engineering [24]. In the reverse engineering stage, the software modernization process analyzes the legacy system (at a lower abstraction level) in order to identify the components of the system and their interrelationships and build one or more representations of the legacy system (at a higher abstraction level). In addition, the metamodel defined by the KDM standard provides a common repository structure that makes it possible to exchange information in the restructuring and forward engineering stages about all existing software artifacts in legacy systems like source codes, databases, user interfaces, business rules, etc. [42].

The representation of the system's knowledge throughout the software modernization stages has four key challenges that must be addressed:

- Legacy systems must be represented from different viewpoints and different abstraction levels where it must be possible to represent all the software artifacts in legacy systems [20]. In addition, the different views of the systems must be linked in order to support the traceability between elements at different views or levels.
- The knowledge embedded in legacy systems must not only be represented in a suitable and accurate way in the reverse engineering stage, but must also be managed throughout the following stages in the whole software modernization process: the restructuring and forward engineering stages.
- It must be possible for different tools to share the recovered knowledge in order to automate the different modernization tasks.
- It must be possible to discover or deduce new implicit knowledge from the explicit knowledge recovered from legacy systems.

All these challenges are met by KDM standard ISO/IEC 19506. This paper presents in detail how the KDM standard can address the challenges presented. Moreover, this paper shows the way in which KDM assists in the entire software modernization process in order to minimize the effects of software erosion on legacy systems.

The remainder of this paper is structured as follows: Section 2 presents a brief history of KDM in order to understand the role that KDM plays in the modernization of legacy systems. Section 3 summarizes works related to the knowledge representation used in software modernization and similar processes. Section 4 presents the elements and features of the metamodel defined by the KDM standard. In addition, this section presents the KDM standard from an ontological perspective and as a common interchange format. Section 5 shows how KDM can be used in ADM-based processes in order to exploit all their benefits. Finally, Section 6 summarizes and discusses the conclusions of this paper.

2. History of KDM

In June 2003, OMG formed a task force to model software artifacts in the context of legacy systems. Initially, the group was called the *Legacy Transformation Task Force*, but soon, the group changed its name to the *Architecture-Driven Modernization Task Force* (ADMTF). In July 2003, the ADMTF issued a software modernization whitepaper [34].

In November 2003, the ADMTF issued the request-for-proposal of the Knowledge Discovery Metamodel (KDM) specification. The objective of this initial metamodel was to provide a comprehensive view of the application structure and data, but it did not represent software below the procedure level. The request-for-proposal stated that the metamodel of the KDM standard:

- represents artifacts of legacy software as entities, relationships and attributes
- includes external artifacts with which software artifacts interact
- supports a variety of platforms and languages
- consists of a platform and language independent core, with extensions where needed
- defines a unified terminology for legacy software artifacts
- describes the physical and logical structure of legacy systems
- can aggregate or modify, i.e. refactor, the physical system structure
- facilitates tracing artifacts from logical structure back to physical structure
- represents behavioral artifacts down to, but not below, the procedural level.

In May 2004, six organizations responded to the request-for-proposal. However, throughout 2004 and 2005 more than 30 organizations from five different countries have collaborated in the development and review of the KDM standard. In May 2006, KDM was adopted by OMG and moved into the finalization stage in the adoption process. In March 2007, OMG presented the recommended specification of KDM 1.0. In April 2007, OMG started ongoing maintenance of the KDM specification.

In January 2009, the recommended specification of KDM 1.1 was published by OMG [42], and in addition, OMG started the revision of that version. Recently, in March 2009, ISO started the adoption process of the KDM specification, which is known as ISO/IEC ADM KDM 19506 [19].

3. Related work

Knowledge management based on models throughout all the stages of software modernization has been widely studied. Several alternative solutions were proposed prior to the KDM standard.

In most cases, the knowledge recovered through reverse engineering techniques is represented and managed in an *ad hoc* way. Zou et al. [55] propose a framework based on a set of heuristic rules for extracting business processes through the static analysis of the source code. The authors of this work also use an algebraic approach in the study of business processes. Favre [13] proposes an MDA-based framework for modernizing legacy systems. This work defines NERUS, an *ad hoc* metamodeling language, to represent metamodels and metamodel transformations. Perez-Castillo et al. [47] propose an ADM-based framework to generate web services from relational database schemas. The authors of this work use a specific metamodel according to the SQL standard to represent the knowledge about

database schemas. Contrary to the previous framework, Vara et al. [51] propose a framework to obtain object relational database schemas from UML models. Maciaszek [25] proposes the meta-architecture called PCMEF (Presentation, Control, Mediator, Entity, and Foundation) to support roundtrip architectural modeling as an alternative solution to MDA. Grau et al. [15] propose PRiM, an *i**-based business process reengineering method for information systems specification. This *ad hoc* method uses the *i** framework [54] which proposes the use of two types of models: (i) the SD (Strategic Dependency) model which represents the intentional concepts of a process; and (ii) the SR (Strategic Rationale) model which represents the rationale behind it. Daga et al. [11] present an ontological approach for recovering legacy business knowledge for modernizing legacy systems. The proposed ontology not only takes the technical aspect of the legacy systems into account, but also considers the business knowledge which resides within the system. Finally, another relevant work is the knowledge management framework proposed by Yang et al. [53]. This framework carries out a hybrid peer-to-peer knowledge management architecture to consider the tradeoff between centralized and decentralized knowledge in a business environment. The approach is based on the transactive memory theory, i.e., the collective memory of small business groups.

Several works opt for the *ad hoc* option because this kind of solution is more suitable to the specific needs in each case. However, the most important problem with *ad hoc* solutions is the standardization problem, since these frameworks or approaches cannot be used in the same way for all kinds of systems, all kinds of domains, all kinds of technologies, etc.

Due to the formalization problem, other authors have used standards to represent and manage the knowledge in the reverse engineering stage. A common standard used in some works is UML. Armour et al. [4] show how to use standard UML models to capture architectural view knowledge from legacy systems. In addition, the authors propose specific UML extensions in order to address the representation of business models. Abdullah et al. [1] propose an extension to UML for knowledge modeling based on the profiling extension mechanism of UML. This UML profile focuses on the knowledge management in knowledge-based systems.

Despite these works, UML was not intended to be used as a metamodel to represent all the artifacts involved in a legacy system that is being modernized. Indeed, UML has several limitations since it does not support the representation of the business view of legacy systems at higher abstraction level. Particularly, another standard called SBVR [40] (Semantics of Business Vocabulary and Business Rules) makes it possible to represent and manage business knowledge. Some works try to recover business logic from legacy systems using this standard. Nelson et al. [32] present a lifecycle of the management of business rules based on the SBVR standard and they compare it to the traditional knowledge management lifecycle. Moreover, Muehlen et al. [31] compare the SBVR with other standards with similar purposes such as SRML (Simple Rule Markup Language) and BPMN (Business Process Model and Notation), and analyze the limitations of the SBVR standard. The main limitation of SBVR is that it does not support the representation of events.

All these proposals are limited to managing the knowledge involved in the reverse engineering stage. On the one hand, some proposals are not adequate due to the fact that they are not standards, neither *de facto* nor *de jure*. And on the other hand, the standard-based proposals focus on the technical view or business view of the legacy system in an isolated way. The KDM standard solves this problem because it is standard, and in addition, KDM makes it possible to represent software artifacts of a technical and business nature at different abstraction levels. For this reason, KDM is the most suitable solution to represent and manage the knowledge involved in any modernization process to minimize the effects of software erosion. The next section presents the KDM standard in detail.

4. Knowledge Discovery Metamodel

The goal of the KDM standard is to define a metamodel to represent all the different legacy software artifacts involved in a legacy information system (e.g. source code, user interfaces, databases, etc.). The metamodel of the KDM standard provides a comprehensive high-level view of the behavior, structure and data of legacy information systems by means of a set of facts. The main purpose of the KDM specification is not the representation of models related strictly to the source code nature such as UML (Unified Modeling Language) [19]. While UML can be used to generate new code in a *top-down* manner, an ADM-based process using KDM starts from the different legacy software artifacts and builds higher-abstraction level models in a *bottom-up* manner through reverse engineering techniques [30].

The KDM specification can be seen from different perspectives. Firstly, KDM can be considered as a metamodel to represent legacy knowledge models. Secondly, most of the KDM specification is a definition of a language- and platform-independent ontology of legacy information systems. Finally, KDM can be also seen as a common interchange format that makes the interoperability between the reverse engineering tools and modernization tools possible. The next subsections show KDM from each perspective.

4.1. KDM as a metamodel

KDM defines an MOF-compliant metamodel, which means that the metamodel of the KDM standard is defined as an Entity-Relationship model according to the MOF (Meta Object Facility) meta-metamodel [37]. Several metamodels standardized by OMG, including UML, CWM (Common Warehouse Metamodel) and SPEM (Software & System Process Engineering Metamodel) [41] use MOF as KDM does.

In order to manage the complexity of KDM, the structure of the metamodel of the KDM standard is divided into several layers representing both physical and logical software artifacts of legacy information systems at different abstraction levels. Furthermore, each abstraction layer separates the knowledge about legacy information systems into various orthogonal concerns that are well-known in software engineering as architecture views [19]. Fig. 1 shows the organization of the metamodel of the KDM standard.

The metamodel of the KDM standard defines four layers: *infrastructure*, *program elements*, *runtime resources* and *abstractions* layer. Each abstraction layer of the metamodel is based on the previous layer. Furthermore, each layer is organized into packages that define a set of metamodel elements whose purpose is to represent a specific independent concern of knowledge related to legacy systems.

4.1.1. Infrastructure layer

The *infrastructure* layer, at the lowest abstraction level, defines a small set of concepts used systematically throughout the entire KDM specification. There are three packages in this layer: *core*, *KDM* and *source*.

4.1.1.1. Core package. The *core* package defines the basic constructs for creating and depicting the metamodel elements in all KDM packages. This means that the *core* package determines the structure of the KDM models and defines the fundamental patterns and constraints implemented by all the other KDM packages.

Therefore, any metamodel element defined in any other package must be a subclass of one of the *core* classes. The two fundamental classes (metamodel elements) of the *core* package are *KDMEntity* and *KDMRelationship*, since KDM is an entity-relationship representation. A KDM entity "is an abstraction of some element of an existing software system that has a distinct, separate existence, a self-contained piece of data that can be referenced as a unit" [19]. Moreover, a KDM relationship "represents some semantic association between elements of an existing software system" [19].

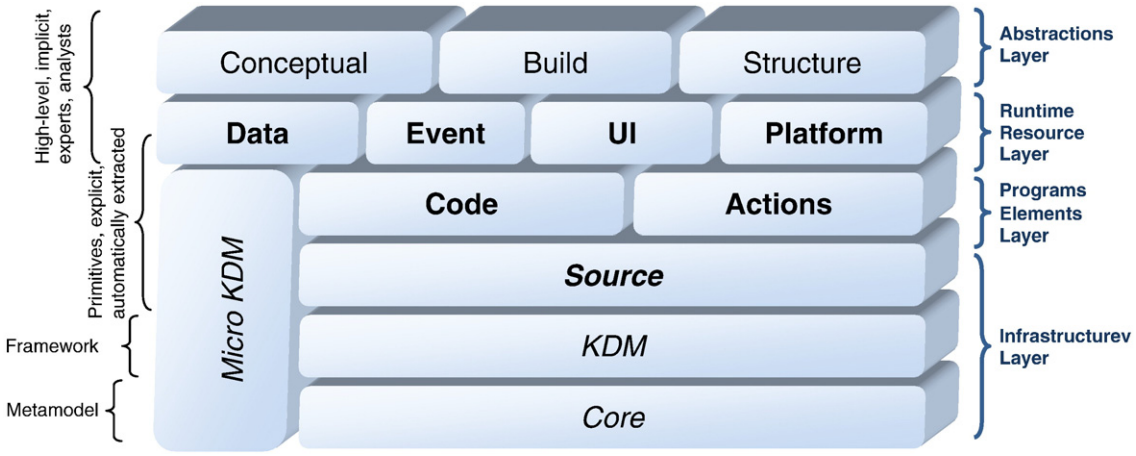

Fig. 1. Layers, packages, and concerns in KDM (Adapted from [19]).

In addition, the *core* package defines the data types that are used to represent all the different models according to packages defined by the KDM specification. Only three types exist: string, integer and boolean, which are subclasses of the MOF *DataType* class. These are not related to the wide variety of datatypes in the system of interest that is being represented by KDM. These others data types are represented by means of particular elements of the *Code* package (cf. Section 4.1.2).

4.1.1.2. KDM package. The *KDM* package also defines the common metamodel elements that constitute the infrastructure for other packages of the metamodel. In addition, the KDM package defines the elements that constitute the framework of each KDM representation. This framework determines the physical structure of a KDM representation (also referred to as KDM instances), i.e., each KDM representation consists of one or more *Segment* elements that own several KDM models.

Each *KDM* package defines some specific kind of KDM model, which addresses a specific concern in the knowledge about the legacy systems. Thus, each specific implementation of KDM might support one or more selected KDM models grouped into *Segment* elements. The KDM implementers are responsible for defining the KDM models used in each case to represent the elements of programming languages, runtime platforms, and any other legacy software artifact.

In addition, the *KDM* package defines the common lightweight extension mechanism to add new metamodel elements in order to extend the semantic in specific KDM representations. This mechanism consists of introducing new types of extended metamodel elements, also known as stereotypes. The stereotypes modify the semantic of a certain element.

The *KDM* package, together with the *core* package, constitutes the so-called KDM framework. These packages do not define any kind of KDM model since they represent the common infrastructure for the rest of the KDM specification.

4.1.1.3. Source package. The *source* package defines the first and most basic model of KDM: the *inventory* model. This model enumerates the physical artifacts of a legacy system (like source files, images, configuration files, resource files, and so on) as KDM entities and defines a traceability mechanism to link those KDM entities and their original representation in the legacy source code, for which the KDM representation was created. Other KDM models in other KDM packages use this mechanism to refer to the physical software artifacts.

Fig. 2 shows the metamodel elements of the inventory model in the *source* package. The *InventoryModel* element represents the root metamodel element in the *source* package. The *InventoryModel* has several *AbstractInventoryElements*, and in turn, it has several *AbstracInventoryRelationships* (both elements are, respectively, subclasses of *KDMEntity* and *KDMRelationship*).

The *AbstractInventoryElement* is specialized into concrete metamodel elements in order to represent different physical artifacts: *BinaryFile*, *ExecutableFile*, *ResourceDescription*, *Configuration* and *Image*. Also, *AbstractInventoryElement* is specialized in container elements such as *Directory* and *Project* that group other inventory elements.

The *source* package offers two mechanisms for linking a KDM element to the corresponding physical artifact, on the one hand, the sequence enumeration of the corresponding fragments of source code (*Segment* element of the *KDM* package) into a KDM representation, and on the other hand, the linkage of a KDM element to a specific region of the source code (*SourceRegion* element) within some physical artifact (*SourceFile* element) by means of a reference element (*SourceRef* element).

Moreover, the inventory elements are interrelated by means of *AbstractInventoryRelationships* (see Fig. 2), which is specialized into *DependsOn* and *InventoryRelationship* elements. *InventoryRelationship* is a generic relationship between any inventory element and any KDM entity in any kind of KDM model. However, the *DependsOn* element represents a relationship between two inventory elements. The *DependsOn* relationship means that an inventory model requires another inventory element during one or more steps of the engineering process. For instance, imagine an executable file that needs a certain configuration file.

4.1.2. Program elements layer

The *program elements* layer offers a broad set of metamodel elements in order to provide a language-independent intermediate representation for various constructs defined by common programming languages. This layer represents implementation level program elements and their associations. This means that the *program elements* layer represents the logical view of a legacy system. This layer has two packages: the *code* package and the *action* package. Both packages define a single model, called the *CodeModel*.

4.1.2.1. Code package. The *code* package defines a set of *CodeItem* elements that represents the common named elements in the source code supported by different programming languages such as data types, classes, procedures, methods, templates and interfaces. Also, the *code* package makes it possible to represent the structural relationships between the code elements.

Fig. 3 shows the metamodel elements of the code model. The *CodeModel* element aggregates several *CodeItem* elements. The *CodeItem* element is specialized into three major metamodel elements: *Module*, *ComputationalObject* and *DataType*.

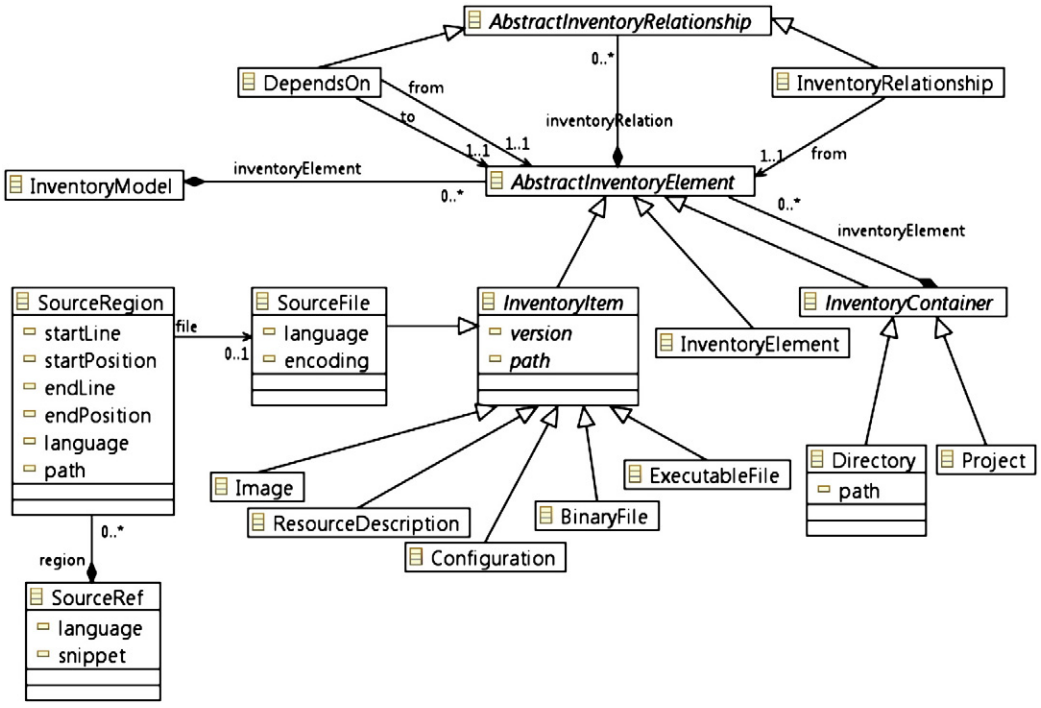

Fig. 2. The inventory model in the KDM source package (adapted from [19]).

- The *Module* element represents "a discrete and identifiable program unit that contains other program elements and may be used as a logical component of the software system" [19]. The *Module* element is specialized into *Package*, *CompilationUnit*, *CodeAssembly*, etc.
- The *ComputationalObject* element and its subclasses represent callable computational objects like *MethodUnits* or *CallableUnits* in general.
- The *DataType* metamodel element is a common superclass that defines the named data items of legacy systems such as global/local variables, record files and formal parameters in callable units. In addition, the data representation in KDM is aligned with the General-Purpose Datatypes (GPD) standard [18]. This standard defines the nomenclature and shared semantics for various datatypes commonly used in programming languages.

Moreover, the code elements that make up the code model can have several code relationships. These relationships are defined by means of the *AbstractCodeRelationship* element and its subclasses: *Imports*, *Extends*, *Implements*, *Include*, among others.

In conclusion, the *code* package provides broad coverage for most common code elements and data types of several programming languages. Nevertheless, the metamodel of the KDM standard also offers the generic extensible mechanism based on stereotypes to

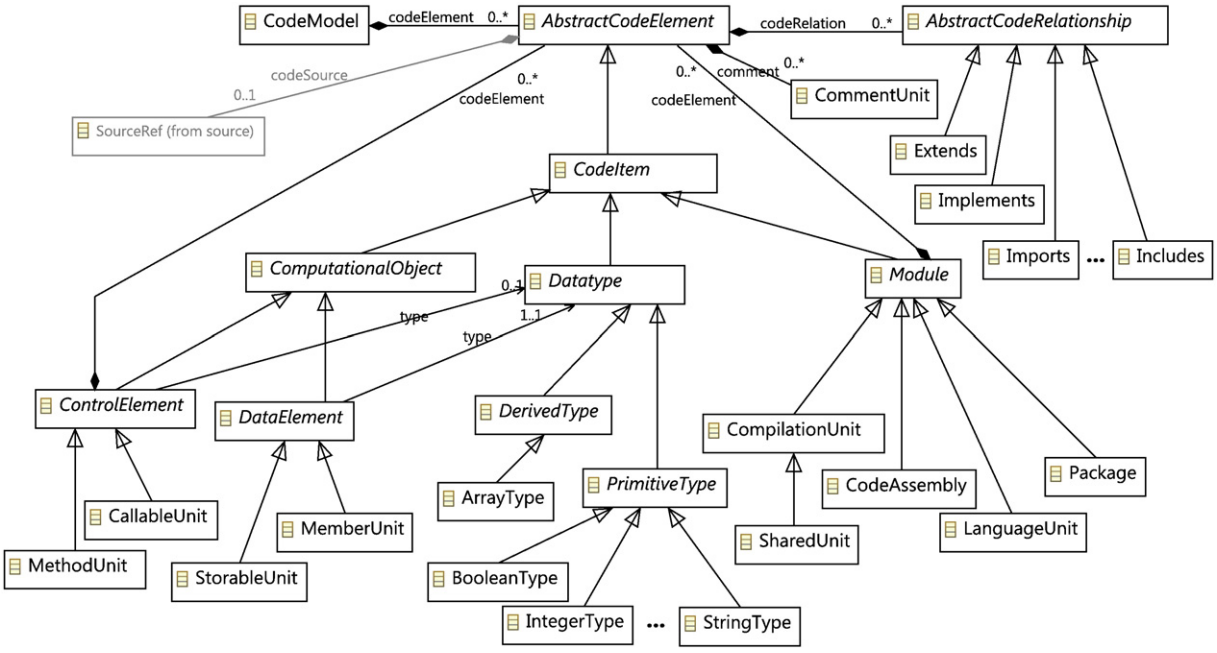

Fig. 3. The code model in the KDM code package (adapted from [19]).

represent those uncommon code fragments not supported by the *code* package.

4.1.2.2. Action package. The *action* package extends the code model by means of more metamodel elements that represent behavior descriptions and control-and-data-flow relationships between code elements. The *action* package adds two key elements: the *ActionElement* and the *AbstractActionRelationship* (see Fig. 4).

The *ActionElement* metamodel element depicts a basic unit of behavior and represents some programming language constructs such as statements, operators and conditions. The feature *kind* in this element specifies the precise semantic of the action according to a finite set of action types defined in the *Micro KDM*, which categorizes the possible actions in legacy information systems by providing additional semantic to each action. Also, the *ActionElements* can be linked to the original representation in legacy code by means of the *SourceRef* element of the *source* package.

The *AbstractActionRelationship* usually represents the use of a name in a statement. This use is done in two ways: this kind of element can represent (i) a control flow between two actions (*ControlFlow* element and its subclasses) or (ii) an association from an action element to a code element in the code model. For example, the specialized element *Calls* refers an *ActionElement* (feature *from*) and also refers a *ControlElement* that typically can be a *CallableUnit* (feature *to*). The *Call* element represents an invocation from a sentence in the legacy code to a callable unit (a method or function). Other specialized elements that are similar to the *Call* element are the *Writes* and *Reads* elements in order to represent the access to data and the *ExceptionFlow* element which represents the anomalous control flows in the source code, among other metamodel elements (see Fig. 4).

4.1.3. Runtime resource layer

The *runtime resource* layer enables the representation of high-value knowledge about legacy systems and their operating environment, i.e., this layer focuses on those things that are not contained within the code itself.

The *runtime resource* layer focuses on representing resources managed by the runtime platform. For this purpose, this layer provides abstract resource actions to manage the resources. Each package in this layer defines specific entities and containers to represent all the resources of the specific concerns of legacy systems. Also, the packages in this layer define specific structural relationships between the resources as well as specific resource actions to depict manipulations of resources related to the metamodel elements of previous KDM layers.

The *runtime resource* layer establishes four packages: *data*, *event*, *UI* and *platform*. In turn, these four packages respectively define four KDM models with the same name.

4.1.3.1. Data package. The *data* package defines the representation of several data organization capabilities. However, this representation is related to the persistent data aspects of a legacy system, since the application data as input/output variables or parameters in callable units are represented through the *code* package. Therefore, this package makes it possible to represent complex data repositories like relational databases, record files and XML schemas and documents.

The *data* package defines *DataModel* in order to represent all the persistent data resources of a specific legacy system (*DataResource* metamodel elements) as well as the actions with those elements (*DataAction* metamodel elements).

The *DataResource* element is specialized into several metamodel elements. Regarding the relational database representation, *DataResource* is specialized into *Catalog* and *RelationSchema* elements. In turn, the *RelationSchema* element can aggregate several *ColumSet* elements, which are specialized into *DataSegment*, *RelationalTable*, *RelationalView*, *RecordFile*, and so on. In addition, the *DataResource* element is specialized into the *IndexElement*, which is also specialized into the *Index*, *UniqueKey* or *ReferenceKey* element.

Also, the *data* package defines a set of data elements regarding the XML schema representation. *AbstractContentElement* represents any XML schema definition and is specialized into *ContentItem*, *ComplexContentItem* and *ContentRestriction*. The *ContentItem* element is specialized into *ContentElement*, *ContentAtribute*, *ContentReference*, etc. *ComplexContentItem* is also specialized to represent *SimpleContent*, *MixedContent*, *SeqContent*, *ChoiceContent*, and so on. Finally, *ContentRestriction* defines XML schema restrictions like *minExclusive*, *length*, and *whitespace*, among others.

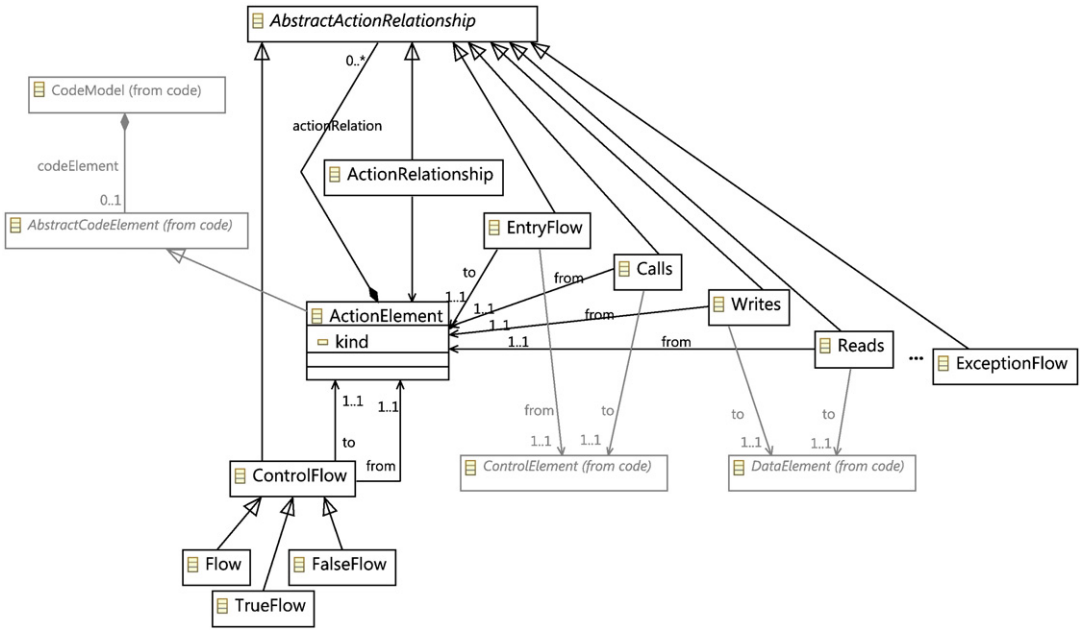

Fig. 4. The code model in the KDM action package (adapted from [19]).

Moreover, the *data* package defines data actions since it extends the *AbstractActionRelationship* element of the *action* package with four new elements. These new action relationships are established from an *ActionElement* in the code model to a *DataResource* element in the data model. The four new relationships are the following: (i) the *ReadColumnSet* element represents a flow of data from the data resource, for instance this element makes it possible to model the *select* SQL queries embedded in the source code; (ii) the *WritesColumnSet* element represents a flow of data to the data resource, for instance it models the *insert* queries; (iii) the *ManageData* element represents the data accesses when there is no flow of data to or from the data resources, thus the nature of the operation is represented by the *kind* attribute related to the *micro KDM* package; and finally (iv) the *HasContent* element represents a structural relationship that defines a linkage between the action element and the data resource.

4.1.3.2. Event package. The *event* package defines a set of metamodel elements for representing state and state transitions caused by events. This package represents two kinds of states: (i) the concrete states that are explicitly supported by specific state-machine based languages, such as CHILL [16]; and (ii) abstract states related to a specific algorithm, resource or user interface.

The *event* package defines the *EventModel* metamodel element that aggregates the set of *EventResources* of a legacy system. The *EventResource* metamodel element is specialized into *State*, *Transition* and *Event* elements. In addition, each *EventResource* element is related to one or more *AbstractEventRelationships* and this element is specialized into two relationships: the *NextState* metamodel element establishes a relationship from a *Transition* to the given *State* after this event-based transition, and *ConsumeEvent* defines a relationship between a *Transition* and the *Event* that triggers that transition.

Moreover, the *event* package extends the *action* package since it defines a set of event actions that can be used with the *EventResources*. These event actions include the *ReadState* element which represents the access to the current state from a specific code element. The *ProducesEvent* element is similar to the *Writes* relationship of the *action* package, since it defines the event that is produced in a concrete code element. Finally the *HasState* element represents a structural relationship and makes it possible to associate an element of an event model with any resource.

4.1.3.3. UI package. The *UI* package provides metamodel elements for representing the resources related to user interface aspects, such as their composition, their sequence of operations and their relationships to the legacy systems.

The *UI* package also defines a specific model through the *UIModel* element in order to statically represent the main components of the user interface of a legacy system. The UI model consists of a set of *UIResource* metamodel elements, and in turn, the *UIResource* element is specialized into *UIDisplay*, *UIField*, *UIEvent*, among others. The UI relationships defined by this package are only two: *UILayout* represents the behavior of the user interface as the sequential flow from one instance of display to another; and *UILayout* establishes a relationship between a portion of a user interface and its layout.

This package provides a set of actions that extends the action elements defined by the *action* package. These action metamodel elements are: *ManageUI*, *ReadsUI* and *WritesUI* and their semantic is very similar to the semantic defined by the *action* package, but these action elements refer to *UIResource* elements.

4.1.3.4. Platform package. The *platform* package defines artifacts related to the runtime platform and environment of a legacy system such as inter-process communication, the use of registries, the management of data, and so on. The platform metamodel elements depict the execution context of the legacy code, since a legacy system is not only determined by the programming language of the source code, but also by the selected runtime platform.

The *platform* package defines the *PlatformModel* element as an aggregation of *AbstractPlatformElements* that is specialized into specific metamodel elements like *Process*, *Thread*, *DataManager*, *FileResource*, *StreamResource*, *PlatformEvent*, *NamingResource*, etc. In addition, this model defines only one platform relationship named *BindTo* that defines an association between two platform resources.

Regarding the actions, the platform model extends the *action* package with four action metamodel elements: *DefinedBy*, *ManagesResource*, *WritesResources* and *ReadsResources*. The semantic of these action elements is very similar to the elements defined by the *action* package, although the actions refer to the platform resources.

4.1.4. Abstractions layer

The *abstractions* layer defines a set of metamodel elements whose purpose is to represent domain-specific knowledge as well as to provide a business overview of legacy information systems. This layer has three packages: *structure*, *conceptual* and *build*; and each package defines its own model.

Throughout the modeling process, the *infrastructure* layer is not used to represent any specific model since the elements of this layer are used by other layers. The following KDM layers, *program elements* layer and *runtime resource* layer are used to model explicit knowledge of the legacy systems. Therefore the models built according to the packages of these layers can be obtained automatically through specific analytic tools. However, the last layer, the *abstractions* layer, is used when the KDM implementer find itself interpreting intent full-on. Thus, this layer focuses on representing the implicit knowledge of the legacy systems. For this purpose, the models built according to this layer are obtained by means of the analysis of the explicit knowledge of the models of the previous layers as well as the help of domain experts.

4.1.4.1. Structure package. The *structure* package represents the architectural components of legacy systems like subsystems, layers, packages and so on. This means that the *structure* package defines the divisions and subdivisions down to the modules defined in the *code* package.

The *structure* package defines the *StructureModel* element that represents the organization of the legacy source code, thus the *StructureModel* aggregates several structure metamodel elements. “Packages are the leaf elements of the structure model and they represent divisions of a system's code parts in discrete and non-overlapping manner” [19]. Then, each *Package* can be separated into *Subsystems*, *Layers* and *Components*, or *ArchitectureViews*.

4.1.4.2. Conceptual package. The *conceptual* package represents the domain-specific information of a legacy system in a conceptual model. The conceptual model can represent a behavior graph with the paths of the application logic and the associated conditions. This behavior graph is built from action elements in the program element layer, although the action elements are enriched with semantics of the domain. Therefore, a process of mining high-level knowledge from the previous low-level KDM models makes it possible to discover additional KDM entities and relationships that represent the *concepts* involved in the system. Nevertheless, the value-added knowledge discovery process is very difficult, and thus this process tends to involve domain experts and application experts.

The *ConceptualModel* element consists of an aggregation of *AbstractConceptual-Elements* (see Fig. 5). This element has a property named *implementation* which specifies the mapping of a specific concept to the *KDMEntities* in low-level models. The *AbstractConceptualElement* is specialized into three metamodel elements: the *ConceptualRole* element represents the role played by a participant in a conceptual association, the *TermUnit* represents a noun concept,

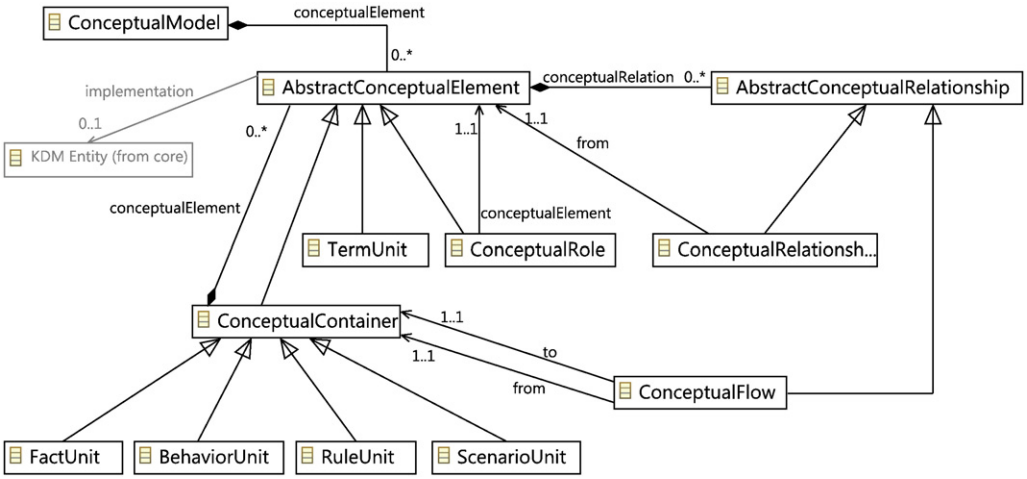

Fig. 5. The KDM conceptual model (adapted from [19]).

and the *ConceptualContainer* represents a container for conceptual entities.

Moreover, the *ConceptualContainer* element is also specialized into *FactUnit*, *RuleUnit*, *BehaviourUnit* and *ScenarioUnit* (see Fig. 5). *FactUnit* associates several conceptual elements and represents some behavior of the legacy system. The *RuleUnit* also associates multiple conceptual entities, but it specifies a condition or a constraint of the behavior of the legacy system. The *BehaviourUnit* element represents a behavior graph with several paths through the application logic. Finally, the *ScenarioUnit* element represents a single path or a set of paths through the behavior graph of the application logic. In addition, the *ConceptualContainer* elements are related by means of *ConceptualFlow* elements, the only kind of relationship defined by the *conceptual* package.

Also, it should be borne in mind that the conceptual package is aligned with the SBVR (*Semantics of Business Vocabulary and Business Rules*) specification [40]. SBVR is an OMG specification related to KDM, which defines a mechanism to discover and represent business rules from information systems. SBVR enables the storage of business rules and business vocabulary in general in an MOF-compliant repository. The KDM conceptual elements that represent the minimal units to represent concepts have a direct correspondence to the elements of SBVR specification: the *TermUnit* corresponds to *Noun* in the SBVR terminology that collectively refers to SBVR *Terms* and SBVR *Names*, the *FactUnit* of KDM corresponds to SBVR *Fact*, and *RuleUnit* corresponds to conditions or constraints in the SBVR specification.

4.1.4.3. Build package. The *build* package defines the artifacts related to the engineering view of a legacy system such as roles, development activities or deliverables generated by the build process. This package defines the *BuildModel* as a model for representing that information. The build model defines a set of build elements such as *Supplier* or *Tool*, and it also provides a set of build resources like *BuildProduct*, *BuildComponent*, *BuildStep*, *BuildDescription*, and so on.

4.2. KDM as an Ontology

KDM can be seen as an ontology [42], since KDM specifies a set of common concepts required for understanding legacy systems and provides infrastructure to support specialized definitions of domain-specific, application-specific and implementation-specific knowledge. Indeed, most of the KDM specification is a definition of a language- and platform-independent ontology of legacy information systems.

The underlying ontology defined in KDM provides the nomenclature needed to represent all the artifacts involved in legacy systems and their relationships. The ontology makes it possible to represent both the structure and behavior of a specific legacy information system at different abstraction levels due to the elements defined in the metamodel of the KDM standard.

With the KDM ontology, the deep semantic integration between software development tools that deal with legacy information systems can take place. At the integration level, the reverse engineering tools and analytic tools need to share the same ontology related to the application. For instance, consider the following three tools: firstly, a reverse engineering tool generates the code model from a source code file; secondly, an analytic tool can derivate the cyclomatic complexity metric of the same source file; and finally, another analytic tool specifies the business rules supported by that source file. These three tools must share the same ontology that involves the concept of *source file* [23].

4.2.1. Micro KDM

In addition, the KDM specification defines the *micro KDM* package in order to refine the semantic defined in the ontology. The *micro KDM* package is mainly applied to the action elements defined in the *action* package. *Micro KDM* is a compliance point for the *action* package of the *code* viewpoint. *Micro KDM* defines a set of compliance rules, additional guidance to building and interpreting extra high-fidelity KDM views, suitable to performing static analysis.

The *micro actions* defined by the catalog of the *micro KDM* package decorate the *macro actions* defined through the *action* elements of the *action* package. In this manner, the control flow and the data flow of the KDM representations are more precise [19].

Each micro action consists of four elements: (i) the action kind, specifying the nature of the operation performed, which is represented by means of the *kind* attribute of the macro action; (ii) outputs, representing the outgoing *Writes* relationships, which represent the result of the action; (iii) inputs, defining the outgoing *Reads* relationships, which represent the arguments of the micro action; and finally (iv) the control part, defining the outgoing control flow relationships.

The *micro KDM* package defines a wide catalog of micro actions, which is divided into ten categories that can be seen in Table 1.

4.3. KDM as a common interchange format

Each KDM representation is a model that represents a specific legacy system. However, a KDM representation is not a model that represents system constraints like a UML model used during the system design stage, but is an intermediate representation to capture

Table 1
Micro actions defined by the *micro KDM* package [19].

Category	Description	Micro Actions
Comparison	This category represents actions related to comparisons between value expressions.	Equals, NotEquals, LessThanOrEqual, LessThan, GreaterThan, GreaterThanOrEqual, Not, And, Or, Xor.
Numerical datatypes	This category represents micro actions related to the primitive numerical datatypes.	Add, Multiply, Negate, Subtract, Divide, Remainder, Successor.
Bitwise operations	This represents actions related to the bitwise operations on primitive datatypes.	BitAnd, BitOr, BitNot, BitXor, LeftShift, RightShift, BitRightShif.
Control	This category represents micro actions that represent the control flow.	Assign, Condition, Call, MethodCall, PtrCall, VirtualCall, Return, Nop, Goto, Throw, Incr, Decr, Swith, Compound.
Data Access	This represents actions related to data access.	FieldSelect, FieldReplace, ChoiceSelect, ChoiceReplace, Ptr, PtrSelect, PtrReplace, ArraySelect, ArrayReplace, MemberSelect, MemberReplace, New, NewArray.
Conversions	This represents the actions related to type conversions	Sizeof, Instanceof, DynCast, TypeCast.
String operations	This represents the actions related to string type operations.	IsEmpty, Head, Tail, Empty, Append.
Set type operations	This represents the actions related to set type operations. The value space of a set element is the set of all subsets of the value space of the element datatype.	IsIn, Subset, Difference, Union, Intersection, Select, IsEmpty, Empty.
Sequence type operations	This represents the actions related to the sequence type operations. The value of a sequence element is an ordered sequence of values of the element datatype.	IsEmpty, Head, Tail, Empty, Append.
Bag type operations	This represents the actions related to bag type operations. The value of a bag element is a collection of instances of values of the element datatype.	IsEmpty, Select, Delete, Empty, Insert, Serialize.

precise knowledge about the existing applications [19]. As a consequence, in addition to a metamodel and an ontology, the KDM specification can be seen as a common interchange format shared for reverse engineering tools, software analysis tools, and any other modernization tool.

The metamodel of the KDM standard defines a common repository structure to exchange the data contained within individual tool models that represent legacy software artifacts. This means that the metamodel defined by the KDM standard provides a common interchange format to enable the interoperability between knowledge discovery tools, services, and respective intermediate representations.

The KDM standard makes it possible to analyze artifacts of legacy information systems and export their representations based on the XMI (XML Metadata Interchange) format [19]. XMI is a model-driven XML (eXtensible Markup Language) integration framework for defining, manipulating and interchanging XML data and objects [38].

Fig. 6 shows an example of an XMI file obtained from a fragment of the legacy source code. The upper part of Fig. 6 represents a fragment of a Java source file that defines a package with two classes. The bottom part of Fig. 6 shows the XMI file that corresponds to the KDM representation of the Java source code. This KDM representation defines a KDM code model using the code and action packages, thus this KDM representation uses the *infrastructure* and *program element* layer. The code model uses the metamodel elements of the *code* package to represent the structure of the source code, i.e., the Java package, the two classes and their methods. In addition, this KDM representation employs the *action* metamodel elements to depict the meaning of each Java sentence.

4.3.1. KDM compliance levels

The metamodel of the KDM standard has a very broad scope since it makes it possible to represent legacy systems with different natures. However, not all KDM capabilities are always applicable to all platforms technologies, applications or programming languages.

Therefore, the KDM specification establishes a set of compliance levels. A compliance level is defined by convention and denotes a specific metamodel subset that ensures the interoperability of a certain kind of tool. Due to the fact that the metamodel of the KDM standard is structured modularly and follows the principle of separation of concerns, the KDM specification can easily define several compliance levels.

The KDM specification defines three compliance levels: level 0 (L0), level 1 (L1) and level 2 (L2).

- Level 0. This compliance level is formed by the following KDM packages: *core*, *kdm*, *source*, *code* and *action*, i.e., the packages of infrastructure layer and program element layer. A tool is L0 compliant when this tool supports all the model elements in those packages. Thus, the L0 compliance level represents the minimal basis for interoperability between different kinds of tools. In fact, to be L1 or L2 compliant, it is first necessary to be L0 compliant.
- Level 1. This compliance level is formed by the remaining KDM packages: *platform*, *data*, *event*, *ui*, *build*, structure and conceptual, which belong to the runtime resource layer and abstractions layer. Also, the elements defined in the *micro kdm* package are considered in this level. To be L1 level compliant, it is not necessary to support all the packages, rather only the packages for the specific domain addressed by the tool. This means that L1 represents a level where tools could be complementary since the tools can focus on different areas of concern.
- Level 2. This level is the union of all the KDM packages of the L1 level. Thus, in order to be L2 compliant, a tool should support the whole metamodel of the KDM standard.

Compliance levels allow tool vendors to select only those parts of the metamodel of the KDM standard that are interesting to a specific tool [19]. When a vendor starts to develop a software modernization tool, it knows *a priori* the parts of the metamodel of the KDM standard that should be supported by the tool in order to be in compliance with the desired level. As a consequence, all the tools in compliance with a specific level are interoperable between each other in that level.

4.3.2. KDM ecosystem

The KDM standard makes it possible to represent and interchange platform- and language-independent models about several artifacts of a legacy system. Due to this fact, the KDM specifications enable the automation and integration of the knowledge discovery processes, since it defines the items that a reverse engineering tool should discover and a software analysis tool can use. Therefore, the KDM standard facilitates knowledge-based integration between different tools.

The KDM standard changes the way in which reverse engineering tools are built and used. Firstly, the traditional reverse engineering

```
package flip;
public class foo {
    public foo() {
    }
    public flip(int i) {
        return i * -1;
    }
}
public class FlipClient {
        public static void main(String[] args) {
        foo f= new foo();
        iFlip g=(iFlip) f;
        f.flip(100);
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment          xmlns:kdm="http://kdm.omg.org/kdm"          xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"      xmlns:action="http://kdm.omg.org/action"
xmlns:code="http://kdm.omg.org/code"                                          xmi:version="2.0"
xsi:schemaLocation="http://kdm.omg.org/action
http://alarcos.esi.uclm.es/per/rpdelcastillo/metamodels/kdm.ecore.xml#//action
http://kdm.omg.org/code
http://alarcos.esi.uclm.es/per/rpdelcastillo/metamodels/kdm.ecore.xml#//code  http://kdm.omg.org/kdm
http://alarcos.esi.uclm.es/per/rpdelcastillo/metamodels/kdm.ecore.xml#//kdm"
name="InterfaceExample.java.kdm">
  <model xsi:type="code:CodeModel" xmi:id="id.0" name="InterfaceExample.java">
    <codeElement xsi:type="code:LanguageUnit" xmi:id="id.1" name="Common Java Datatypes" />
    <codeElement xsi:type="code:Package" xmi:id="id.2" name="flip">
      <codeElement xsi:type="code:ClassUnit" xmi:id="id.3" name="foo">
        <codeElement xsi:type="code:Signature" xmi:id="id.4" name="foo" />
        <codeElement xsi:type="code:MethodUnit" xmi:id="id.5" kind="constructor" name="foo">
          <codeElement xsi:type="code:Signature" xmi:id="id.6" name="foo" />
        </codeElement>
        <codeElement xsi:type="code:MethodUnit" xmi:id="id.7" kind="abstract" name="flip">
          <codeElement xsi:type="code:Signature" xmi:id="id.8" name="flip">
            <parameterUnit xmi:id="id.9" name="i" />
          </codeElement>
        </codeElement>
      </codeElement>
      <codeElement xsi:type="code:ClassUnit" xmi:id="id.10" name="FlipClient">
        <codeElement xsi:type="code:Signature" xmi:id="id.11" name="FlipClient" />
        <codeElement xsi:type="code:MethodUnit" xmi:id="id.12" name="main">
          <codeElement xsi:type="code:Signature" xmi:id="id.13" name="main">
            <parameterUnit xmi:id="id.14" name="args" />
          </codeElement>
          <codeElement xmi:id="id.15" xsi:type="code:StorableUnit" name="f" kind="local" />
          <entryFlow xmi:id="id.16" from="id.12" to="id.17" />
          <codeElement xsi:type="action:ActionElement" xmi:id="id.17" name="a1" kind="Assign">
            <actionRelation xsi:type="action:Reads" xmi:id="id.18" from="id.17" to="id.15" />
            <actionRelation xsi:type="action:Flow" xmi:id="id.19" from="id.17" to="id.22" />
          </codeElement>
          <codeElement xmi:id="id.20" xsi:type="code:StorableUnit" name="g" kind="local" />
          <codeElement xsi:type="action:ActionElement" xmi:id="id.22" name="a2" kind="MethodCall"/>
        </codeElement>
      </codeElement>
    </codeElement>
  </model>
</kdm:Segment>
```

Fig. 6. Example of KDM representation based on XMI from a fragment of Java source code.

tools have been built as silos[1] where each tool focuses on recovering and analyzing different *proprietary* contents in an isolated way (see Fig. 7 left side). For instance, suppose that we carry out a knowledge discovery process of a legacy enterprise application. We would use a reverse engineering tool for the source code and another tool for the legacy database. As a consequence, at the end of the process we would have two proprietary and independent models: a source code model and a database model. These models must be also analyzed independently through two different analytic tools because the current elements do not match in their representations.

Secondly, the KDM standard makes it possible to build reverse engineering tools in a KDM ecosystem. This means that the reverse engineering tools can recover different knowledge related to different artifacts, but this knowledge is represented and managed in an integrated and standardized manner through KDM (see Fig. 7 right side). In this way, the analytic tools can analyze the shared KDM repository and generate the new knowledge. Thus, the advantages derived from the integration of this kind of tool are the following:

- It makes it possible to deal with a diversity of implementation languages and runtime platforms.
- In the future, more analytic tools can be plugged into the KDM models homogeneously to generate more and more valuable knowledge. Thus, it facilitates the complete integration of the different views of the legacy system.
- It makes it possible to understand the "as is" architecture of the legacy system and manages changes from an architectural perspective.
- It makes it possible to combine static analysis with dynamic analysis and metrics for a great amount of software artifacts.

5. Using KDM to modernize legacy systems

The software modernization approach, and particularly ADM, is a mechanism to deal with negative software erosion effects in legacy

[1] Silo: in the context of KDM, a silo is a software solution that consists of several applications used in a sequential way to reach a common goal. The applications of the solution use a common but proprietary format to interchange the shared knowledge throughout the silo. Thus, other similar applications cannot be integrated beyond the solution.

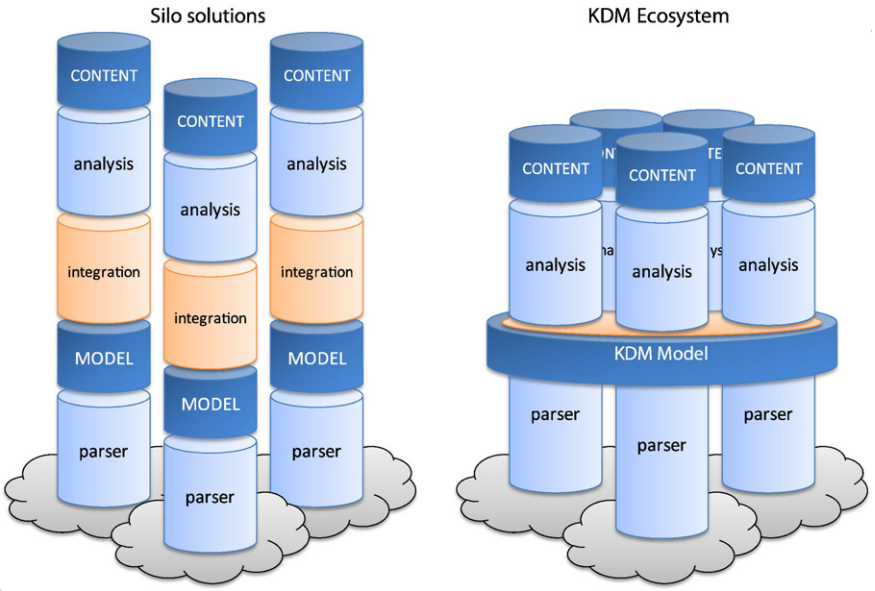

Fig. 7. Software archeology tools: silo solutions (left side) and KDM ecosystem (right side).

systems. According to [35], ADM is the concept of modernizing existing systems with a focus on all aspects of the current system architecture and the ability to transform current architectures to target architectures. Thus, an ADM-based process understands and evolves existing software assets and restores the value of existing applications.

ADM is based on traditional reengineering processes, but it takes *model-driven* principles into account. The horseshoe reengineering model has been adapted to ADM and it is known as the horseshoe modernization model [24]. This model is divided into three stages (see Fig. 8):

- Reverse engineering represents the left side of the horseshoe. It analyzes the legacy system in order to identify the components of the system and their interrelationships. In turn, the reverse engineering stage builds one or more representations of the legacy system at a higher level of abstraction.
- Restructuring represents the curve of the horseshoe since this stage takes the previous system's representation and transforms it into another at the same abstraction level. This stage preserves the external behavior of the legacy system.
- Forward engineering represents the right side of the horseshoe because it generates physical implementations of the target system at a low abstraction level from the previous restructured representation of the system.

KDM make it possible to represent the legacy knowledge extracted from different software artifacts in legacy systems. The software artifacts are analyzed and the KDM models can be built directly according to the metamodel of the KDM standard. However, the KDM models can be also built in two steps: (i) firstly, the recovered information from the software artifacts is used to obtain a PSM model according to a specific metamodel (for example, a code model according to a Java metamodel); and then, (ii) the KDM model, which can be considered as a PIM model, is obtained from the earlier specific model by means of model transformations.

From the point of view of KDM, the reverse engineering stage is probably the most important stage in the horseshoe modernization process. This is due to the fact that this activity conditions the abstraction level achieved in each kind of modernization process and therefore the resources and possibilities to restructure after the legacy systems. The reverse engineering stage is the main activity especially

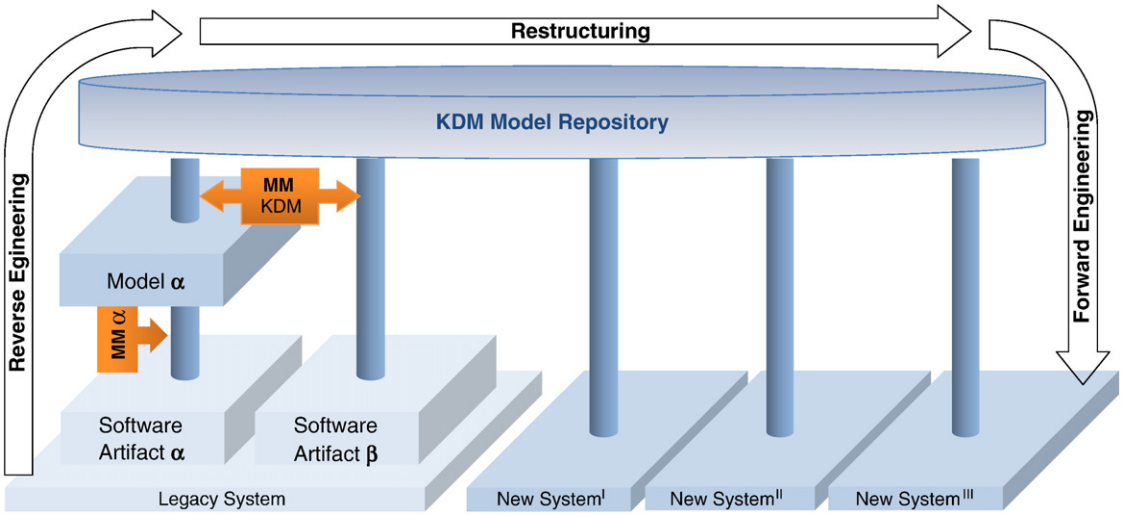

Fig. 8. The role of KDM within horseshoe modernization model.

when a specific modernization process wishes to obtain business process models, business rules and any other business semantics from the legacy systems, i.e., the higher abstraction level. In this kind of modernization process there is a large conceptual gap between business processes and legacy systems that needs to be gradually reduced. KDM reduces this gap due to the fact that it is organized into several layers at different abstraction levels. Thus, KDM makes it possible to carry out *endogenous transformations*[2] from models in lower layers to models in higher layers of the KDM structure. Therefore, specific business knowledge can be extracted directly from the legacy system, and then the implicit knowledge at higher abstraction levels is inferred or deduced from the prior knowledge.

KDM also assists in the restructuring and forward stages of the modernization process. In this case, KDM acts as common interchange format for several restructuring tools that can modify, restructure and improve the recovered KDM models. In addition, KDM offers a KDM model repository according to the KDM ecosystem. Thus, the forward engineering stage can be aided by KDM because the KDM model repository is also shared by forward engineering tools such as generative tools. Automatic Code Generation tools can use KDM models at the begining of the modernization process to generate the respective source code and other software artifacts regarding the target information system by using forward engineering techniques.

5.1. ADM standards

ADM not only defines KDM, but also defines an entire set of standards that complement the KDM standard in order to support all of the throughout the whole ADM-based process. The other standards are being developed around KDM, although some of them are still in the approval or development stage [43]:

- ASTM (Abstract Syntax Tree Metamodel) represents knowledge discovered in legacy systems at the intermediate level of abstraction under KDM by means of abstract syntax tree models.
- SMM (Software Metrics Package) defines a metamodel for representing measurement information related to software, its operation, and its design.
- ADM Pattern Recognition specification aims to be a standard to facilitate the examination of structural metadata with the intent of deriving patterns about legacy systems. It is an on-going activity on the roadmap of the ADM Task Force, leading to a request for proposals in the future.
- ADM Visualization specification focuses on the different ways to show application metadata stored within the KDM models. It is an on-going activity on the roadmap of the ADM Task Force.
- ADM Refactoring specification defines ways in which the KDM specification can be used to refactor applications by means of structuring, rationalizing, modularizing, and so on. It is also an on-going activity on the roadmap of the ADM Task Force.
- ADM Transformation specification seeks to define mappings between KDM, ASTM and target models. It is an on-going activity on the roadmap of the ADM Task Force.

Fig. 9 shows the set of ADM standards contextualized in the horseshoe modernization model as well as other OMG business modeling standards related to ADM-based processes. The KDM standard (ISO/IEC 19506) is the core for the rest of the standards, although KDM is itself sufficient to address software modernization processes that minimize the negative effects of software erosion.

5.2. Using ADM/KDM in research and industrial projects

The KDM standard and the ADM paradigm in general have been applied in several software modernization scenarios. Two important European projects, *MOMOCS* [29] and *ModelWare* [27], aim to lead the successful adoption of ADM by the industry. While the *ModelWare* project focuses on most parts of model-driven development, the *MOMOCS* project particularly addresses the activities and techniques of the software modernization approach.

In addition, there are several companies that use the KDM standard. *KDM Analytics* was the first company to provide a reference implementation of KDM known as *KDM SDK 2.0* [22]. This implementation is an *Eclipse™* EMF plug-in which provides a set of tools for working with KDM. *KDM SDK 2.0* is an adoption kit that facilitates the design of mappings from proprietary internal representation in the metamodel of the KDM standard and jumpstarts development of KDM tools. In 2009, *KDM Analytics* provided the *KDM Workbench* tool, which has *Eclipse™* plug-ins for C, C++, Java, and Cobol.

Other companies have also used KDM in some projects. For example, Benchmark Consulting Services Inc. addresses the modernization of mainframe Cobol systems using KDM [6]. Adaptive Inc. also uses the KDM standard together with other OMG standards in order to evolve legacy information systems [2]. MicroFocus provides a bridge from their software modernization tool for Cobol, SQL and CICS to KDM [26]. Allen Systems Group Inc. proposes integrated modernization environment (IME) supported by a multi-technology repository based on KDM [3]. *Government Computer News* presents a technical report that shows why organizations that provide tools and services for software development lifecycle may need KDM [14].

MoDisco is another important tool that consists of an *Eclipse™* plug-in for model-driven reverse engineering. With *MoDisco*, the practical extraction of models can be done from various kinds of legacy systems. In addition, *MoDisco* proposes a generic and extensible metamodel-driven approach to model discovery [28].

Moreover, *MARBLE*, a non-commercial tool based on the *Eclipse™* platform, can also be used to recover business processes from legacy systems in order to carry out business modernization processes [45]. This tool obtains business processes through three transformations: firstly, the tool recovers PSM models from different legacy software artifacts; secondly, the tool integrates those models into a single PIM model according to the metamodel of the KDM standard; and finally, *MARBLE* recovers a business process model by applying a set of business patterns in the KDM model [46]. MARBLE can be seen as a set of tools working against the same KDM repository.

Ulrich and Newcomb have collected various ADM-based case studies [50]. Some of them consist of real world case studies involving the automated transformation of information systems, from their legacy source code, into modernized languages and platforms in which KDM is used to represent and manage all this information. For example, DelaPeyronnie et al. [12] present an ADM-based project for modernizing the EATMS system, an air traffic management system used in 280 airports worldwide. This project was carried out in a platform migration scenario, since the main objective was the transformation of the legacy system from the *Ada* language into the high-performance, real-time *Java* language. Another successful case study was the modernization project presented by Barbier et al. [5], which focuses on how PIM models can be automatically generated from Cobol-based legacy systems.

Finally, KDM is being used in a wide variety of projects in the software assurance field, as a vendor-neutral standard to formalize machine-readable content for assurance, vulnerability patterns, to perform static analysis directly on the KDM representation of code, as

[2] Endogenous transformation. Model transformations can be classified into endogenous and exogenous transformations. While exogenous transformations involve two models represented according to two different metamodels, endogenous transformations are performed between two models represented according to the same metamodel.

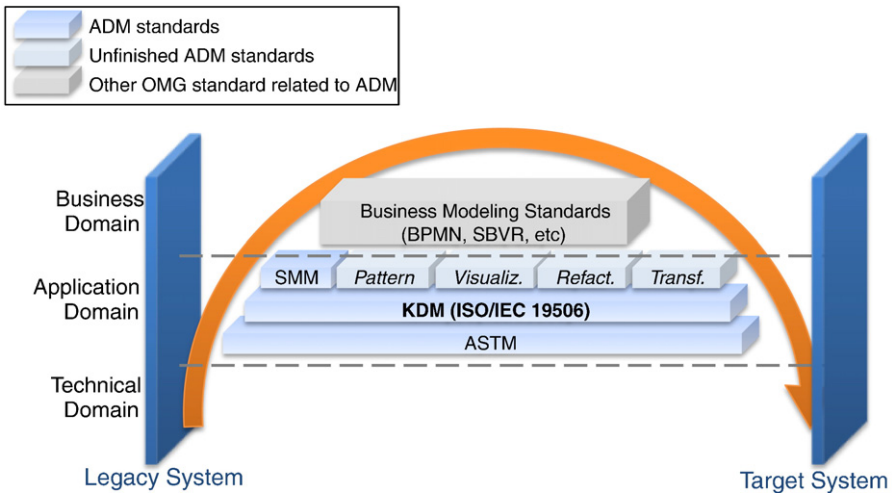

Fig. 9. OMG ADM Standards within the horseshoe modernization model.

well as for automatically generating test cases in multiple programming languages for the security vulnerability detection tools.

6. Conclusions

ADM advocates carrying out reengineering processes takes MDA principles into account. ADM enables the evolution of the legacy systems eradicating (or at least minimizing) the negative effects of the software erosion phenomenon. The main advantage of ADM with respect to traditional evolutionary maintenance processes like reengineering is that it formalizes and standardizes evolutionary maintenance processes. With this advantage, the modernization process can be automatic, reusable and repeatable.

This paper has shown the KDM standard that belongs to the set of standards to modernize legacy information systems defined by OMG within the ADM Task Force. KDM is the first standard defined by the ADM Task Force and it is also the first standard ratified by the ISO/IEC organization as ISO/IEC 19506. The KDM standard makes it possible to represent and manage the knowledge recovered throughout all the modernization stages: reverse engineering, restructuring and forward engineering.

The KDM standard defines a metamodel that specifies a broad set of metamodel elements depicting all the different software artifacts involved in legacy information systems. The metamodel defined by the KDM standard is organized into four abstraction layers, and thus it makes the representation of different models possible that depict knowledge embedded in legacy artifacts at different abstraction levels.

Furthermore, KDM can be seen as a standard that defines an ontology that includes all the elements involved in legacy systems. KDM defines the *micro KDM* package to refine the semantic of some metamodel elements defined in the metamodel of the KDM standard.

The underlying ontology together with the metamodel defined by the KDM standard provides a common semantic for artifacts of several kinds of legacy systems. As a consequence, the KDM standard can be also used as a common interchange format by modernization tools. The different KDM models can be represented as XMI files with a well-defined format according to the metamodel of the KDM standard. These files share the information of the KDM models between different tools. For instance, some reverse engineering tools can recover KDM models from the legacy systems, and then, other analytic or refactoring tools can use those KDM models.

Therefore, the KDM standard solves the four challenges introduced at the beginning of this paper about knowledge management in software modernization processes:

- The layer organization of the metamodel of the KDM standard enables the knowledge representation of all the software artifacts (source code, databases, user interfaces, business rules, etc.) from different viewpoints and different abstraction levels.
- In addition, the KDM standard makes it possible to manage the legacy knowledge recovered from a legacy system throughout all the stages in an ADM-based process due to the fact that ADM provides a set of complementary standards.
- The knowledge recovered from legacy systems and represented in KDM models can be shared by different modernization tools according to the KDM ecosystem.
- Finally, due to the fact that KDM is divided into different abstraction layers, it is able to establish endogenous model transformations between layers that represent the explicit knowledge and layers that represent the implicit knowledge of legacy systems.

Acknowledgments

This work was supported by the Spanish FPU Programme, and by the R+D projects ALTAMIRA (JCCM, PII2I09-0106-2463), PEGASO/MAGO (MICIN and FEDER, TIN2009-13718-C02-01) and MEDUSAS (CDTI (MICINN), IDI-20090557).

References

[1] M.S. Abdullah, et al., Knowledge modelling using uml profile for knowledge-based systems development, in Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering, Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies, IOS Press, 2007, pp. 74–89.
[2] Adaptive Inc. *Adaptive standards* 2008, [cited 2009 18/12/2009]; Available from: http://www.adaptive.com/standards.html.
[3] Allen Systems Group I, Presentation to Address Integrated Modernization Environment (IME) Issues and Market-Leading ASG-becubic™ Multi-Technology Repository, 2009, [cited 2009 18/12/2009]; Available from: http://www.asg.com/newsroom/pr_details.asp?id=136.
[4] F. Armour, et al., A UML-driven enterprise architecture case study, Proceedings of the 36th Annual Hawaii International Conference on System Sciences, IEEE Computer Society, 2003.
[5] F. Barbier, et al., Model-driven reverse engineering of COBOL-based applications, in: W.M. Ulrich, P.H. Newcomb (Eds.), Information Systems Transformation. Architecture Driven Modernization Case Studies, Morgan Kauffman, Burlington, MA, 2010, pp. 283–299.

[6] Benchmark Consulting Services I, Mainframe COBOL Application Modernization: A Revolutionary Approach to Evolving Legacy System. [cited 2009 18/12/2009]; Available from, http://www.benchmarkconsulting.com/AboutUs_nr_ct.html 2007.
[7] K.H. Bennett, V.T. Rajlich, Software maintenance and evolution: a roadmap, Proceedings of the Conference on the Future of Software Engineering, ACM, Limerick, Ireland, 2000.
[8] A. Bianchi, et al., Iterative reengineering of legacy systems, IEEE Transactions on Software Engineering 29 (3) (2003) 225–241.
[9] G. Canfora, M. Di Penta, New frontiers of reverse engineering, 2007 Future of Software Engineering, IEEE Computer Society, 2007.
[10] E.J. Chikofsky, J.H. Cross, Reverse engineering and design recovery: a taxonomy, IEEE Software 7 (1) (1990) 13–17.
[11] A. Daga, et al., An ontological approach for recovering legacy business content, Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05), IEEE Computer Society, 2005, pp. 224–232, Track 8 - Volume 08.
[12] J. DelaPeyronnie, et al., Modernization of the Eurocat Air Traffic Management System (EATMS), in: W.M. Ulrich, P.H. Newcomb (Eds.), Information Systems Transformation. Architecture Driven Modernization Case Studies, Morgan Kauffman, Burlington, MA, 2010, pp. 91–131.
[13] L. Favre, Modernizing software & system engineering processes, International Conference on Systems Engineering, IEEE Computer Society, 2008.
[14] Government Computer News, All for one, but not one for all, 2007. Mar 18, 2007; Available from: http://gcn.com/articles/2007/03/18/all-for-one-but-not-one-for-all.aspx.
[15] G. Grau, X. Franch, N.A.M. Maiden, PRiM: An i*-based process reengineering method for information systems specification, Inf. Softw. Technol. 50 (1–2) (2008) 76–100.
[16] ISO/IEC, ISO/IEC 9496:2003, CHILL - The ITU-T programming language, 2003, http://www.iso.org/iso/catalogue_detail.htm?csnumber=34084, 2003, ISO/IEC. p. 221.
[17] ISO/IEC, ISO/IEC 14764:2006, Software engineering—software life cycle processes—maintenance http://www.iso.org/iso/catalogue_detail.htm?csnumber=39064 2006, ISO/IEC.
[18] ISO/IEC, ISO/IEC 11404:2007, General-Purpose Datatypes (GPD), http://www.iso.org/iso/catalogue_detail.htm?csnumber=39479, 2007, ISO/IEC. p. 96.
[19] ISO/IEC, ISO/IEC DIS 19506. Knowledge Discovery Meta-model (KDM), v1.1 (Architecture-Driven Modernization), http://www.iso.org/iso/catalogue_detail.1128htm?csnumber=32625, 2009, ISO/IEC. p. 302.
[20] J. Jokinen, H.-M. Järvinen, T. Mikkonen, Incremental introduction of behaviors with static software architecture, Comput. Stand. Interfaces 25 (3) (2003) 215–222.
[21] R. Kazman, S.G. Woods, S.J. Carrière, Requirements for integrating software architecture and reengineering models: CORUM II, Proceedings of the Working Conference on Reverse Engineering (WCRE'98), IEEE Computer Society, 1998.
[22] KDM Analytics, *Knowledge Discovery Metamodel (KDM) Software Development Kit 2.0 Eclipse Plugin*KDM Analytics, Inc, http://www.kdmanalytics.com/kdmsdk/KDMSDK_brochure.pdf 2008.
[23] KDM Analytics, *Why Knowledge Discovery Metamodel?* KDM Analytics, Inc, 2008, http://kdmanalytics.com/kdm/ why_kdm.php.
[24] V. Khusidman, W. Ulrich, Architecture-Driven Modernization: Transforming the enterprise. DRAFT V.5 http://www.omg.org/docs/admtf/07-12-01.pdf 2007. OMG. p. 7.
[25] L.A. Maciaszek, Roundtrip architectural modeling, Proceedings of the 2nd Asia-Pacific Conference on Conceptual modelling, 43, Australian Computer Society, Inc, Newcastle, New South Wales, Australia, 2005, pp. 17–23.
[26] Micro Focus, Modernization Workbench™, http://www.microfocus.com/products/modernizationworkbench/ 2009.
[27] ModelWare. MODELWARE. Project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002–2006) 2006 08-2009]; Available from: http://www.modelware-ist.org/.
[28] MoDisco, KDM-to-UML2 converter, http://www.eclipse.org/gmt/modisco/toolBox/KDMtoUML2Converter/.
MoDisco, Eclipse Incubation Project. IST European MODELPLEX project (MODELing solution for comPLEX software systems, FP6-IP 34081), 2008.
[29] MOMOCS, MOdel driven MOdernisation of Complex Systems is an EU-Project, http://www.momocs.org/ 2008; [cited 2008; Available from: http://www.momocs.org/.
[30] B. Moyer, Software archeology. Modernizing old systems, Embedded Technol. J. 1 (2009) 1–4.
[31] M.z. Muehlen, M. Indulska, G. Kamp, Business process and business rule modeling languages for compliance management: a representational analysis, Tutorials, Posters, Panels and Industrial Contributions at the 26th International Conference on Conceptual Modeling, 83, Australian Computer Society, Inc, Auckland, New Zealand, 2007, pp. 127–132.
[32] M.L. Nelson, R.L. Rariden, R. Sen, A lifecycle approach towards business rules management, International Conference on System Sciences, IEEE Computer Society, Hawaii, 2008.
[33] P. Newcomb, Architecture-Driven Modernization (ADM), Proceedings of the 12th Working Conference on Reverse Engineering, IEEE Computer Society, 2005.
[34] OMG, Why Do We Need Standards for the Modernization of Existing Systems?, 2003, OMG ADM Task Force.
[35] OMG, ADM Glossary of Definitions and Terms, http://adm.omg.org/ADM_Glossary_Spreadsheet_pdf.pdf 2006, OMG. p. 34.
[36] OMG, Architecture-Driven Modernization Roadmap, Object Management Group, 2006.
[37] OMG, Meta Object Facility (MOF™) Version 2.0 http://www.omg.org/spec/MOF/2.0/PDF/ 2006, OMG.
[38] OMG, XML Metadata Interchange. MOF 2.0/XMI Mapping, v2.1.1 http://www.omg.org/spec/XMI/2.1.1/PDF 2007, OMG.
[39] OMG, QVT. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification http://www.omg.org/spec/QVT/1.0/PDF 2008, OMG.
[40] OMG, Semantics of Business Vocabulary and Business Rules (SBVR), v1.0. OMG Available Specification http://www.omg.org/docs/formal/08-01-02.pdf, 2008, OMG p. 392.
[41] OMG, SPEM. Software & Systems Process Engineering Meta-Model, v 2.0 Specification http://www.omg.org/cgi-bin/doc?formal/2008-04-01 2008, OMG.
[42] OMG, Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM), v1.1, http://www.omg.org/spec/KDM/1.1/PDF/2009, OMG. p. 308.
[43] OMG, Architecture-Driven Modernization Standards Roadmap. 2009 February 2009 29/10/2009]; Available from:http://adm.omg.org/ADMTF%20Roadmap.pdf.
[44] B. Paradauskas, A. Laurikaitis, Business knowledge extraction from legacy information systems, Journal of Information Technology and Control 35 (3) (2006) 214–221.
[45] R. Pérez-Castillo, et al., MARBLE: A Modernization Approach for Recovering Business Processes from Legacy Systems, International Workshop on Reverse Engineering Models from Software Artifacts (REM'09), Simula Research Laboratory Reports, Lille, France, 2009, pp. 17–20.
[46] R. Pérez-Castillo, et al., Business process patterns for software archeology, 25th Annual ACM Symposium on Applied Computing (SAC'10), ACM, Sierre, Switzerland, 2010, pp. 165–166.
[47] R. Pérez-Castillo, et al., PRECISO: a reengineering process and a tool for database modernisation through web services in 24th Annual ACM Symposium on Applied Computing (SAC'09), 2009, pp. 2126–2133, Hawaii, USA.
[48] H.M. Sneed, Estimating the costs of a reengineering project, Proceedings of the 12th Working Conference on Reverse Engineering, IEEE Computer Society, 2005, pp. 111–119.
[49] W.M. Ulrich, Legacy Systems: Transformation Strategies, Prentice Hall, 2002, p. 448.
[50] W.M. Ulrich, P.H. Newcomb, Information systems transformation, Architecture Driven Modernization Case Studies, Morgan Kauffman, Burlington, MA, 2010, p. 429.
[51] J. Vara, et al., Supporting model-driven development of object-relational database schemas: a case study, in: R. Paige (Ed.), Theory and Practice of Model Transformations, Heidelberg, Springer Berlin, 2009, pp. 181–196.
[52] G. Visaggio, Ageing of a data-intensive legacy system: symptoms and remedies, Journal of Software Maintenance 13 (5) (2001) 281–308.
[53] H.-L. Yang, H.-C. Ho, Emergent standard of knowledge management: hybrid peer-to-peer knowledge management, Journal of Computer Standards and Interfaces 29 (4) (2007) 413–422.
[54] You, E., Modelling Strategic Relationships for Process Reengineering, PhD. thesis. 1995, University of Toronto.
[55] Y. Zou, et al., Model-Driven Business Process Recovery, Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004), IEEE Computer Society, 2004, pp. 224–233.

Ricardo Pérez-Castillo holds the MSc degree in Computer Science from the University of Castilla-La Mancha, and he is currently a PhD student in Computer Science. He Works in Alarcos Research Group at the University of Castilla-La Mancha. His research intererests include architecture-driven modernization, model-driven development and business process recovery. Contact him at Escuela Superior de Informática, Paseo de la Universidad 4, 13071-Ciudad Real, Spain; Ricardo.PdelCastillo@uclm.es.

Ignacio García-Rodriguez de Guzmán is assistant professor at the University of Castilla-La Mancha and belongs to the Alarcos Research Group at the UCLM. He holds the PhD degree in Computer Science from the University of Castilla-La Mancha. His research interests include software maintenance, software modernization and service-oriented architecture. Contact him at Escuela Superior de Informática, Paseo de la Universidad 4, 13071-Ciudad Real, Spain; Ignacio.GRodriguez@uclm.es.

Mario Piattini is full professor at the UCLM. His research interests include software quality, metrics and maintenance. He holds the PhD degree in Computer Science from the Technical University of Madrid, and leads the Alarcos Research Group at the Universidad de Castilla-La Mancha. He is CISA, CISM e CGEIT by ISACA. Contact him at Escuela Superior de Informática, Paseo de la Universidad 4, 13071-Ciudad Real, Spain; Mario.Piattini@uclm.es.