Bridging KDM and ASTM for Model-Driven Software Modernization

Gaëtan Deltombe
Netfective Technology Software
32, avenue Léonard de Vinci
33600 – Pessac, France
g.deltombe@netfective.com

Olivier Le Goaer, Franck Barbier
University of Pau
Avenue de l'université
64000 – Pau, France
{olivier.legoaer, franck.barbier}@univ-pau.fr

Abstract

Standardizing software modernization techniques has lead to the KDM (Knowledge Discovery Metamodel). This metamodel represents several application aspects (code, architecture, etc.), while transforming them into renewed versions. On the other hand, ASTM (the Abstract Syntax Tree Metamodel) has been recently released. It focuses on the parsing of text-based files written in a given language. In practice, KDM and ASTM are intended to be used jointly when modeling source code with formal links to other software features like components, user interfaces, etc. However, the link between ASTM and KDM is often fuzzy, or even unestablished since KDM is in charge of synthesizing all captured software artifacts. This has negative effects on the attainable level of automation and on the completeness of a software modernization project. To overcome this limitation, this paper introduces SMARTBRIDGE *as a means to reconcile both standards.*

1. Introduction

Modernization is at the heart of many software organizations that seek to migrate from obsolete or aging languages and platforms to more modern environments. Modernization projects have historically focused on transforming technical architectures; that is, moving from one platform to another and / or from one language to another [2]. This is achieved through code translation or various refactoring exercises such as restructuring, data definition rationalization, re-modularization or user interface replacement.

Meanwhile, model-driven development (MDD) is gaining increasing acceptance; mainly because it raises the level of abstraction and automation in software construction. MDD techniques, such as metamodeling and model transformation, not only apply to the creation of new software systems but also can be used to help existing systems evolve [7, 1]. These techniques can help reduce software evolution costs by automating many basic activities, including code manipulation.

There is currently great activity addressing model-driven modernization issues, for which the OMG's task force on modernization [16] plays a major role. It aims at making precise inventories of various kinds of legacy artifacts in order to propose more or less automatic model-driven migrations of applications by means of interoperable tools. This way, all discovered artifacts are considered as full-fledged models. This is an important shift compared to classical approaches, which do not consider abstract representations as perennial and reusable assets.

The context of this paper is rooted in the research work conducted by the European REMICS (www.remics.eu) project [11]. This project seeks an end-to-end model processing chain to transform legacy applications/information systems into services in the Cloud. Several modeling languages are operated in the project, within both reverse and forward engineering activities. To define and support this chain in tools, modeling language semantic gaps must be fulfilled. More specifically, the reverse engineering activity must be committed to go from the source code towards technologically-neutral UML models, which can then be consumed by a wide range of third-party MDD tools. The forward engineering activity starts from UML models and next uses SoaML (www.soaml.org) and CloudML (a forthcoming OMG standard that is currently specified within REMICS). So, between reverse and forward, a suite of models conforming to (i.e., instances of) OMG standard metamodels are involved, each aiming at representing the initial source code at different levels of abstraction, along with various refinements/quality levels [8]. The general idea is that a portfolio of metamodels and transformation chains are pre-implemented in a tool to support an intelligible seamless reverse and forward engineering process. Because of its industrial impact, REMICS complies to worldwide standards. Nonetheless, these standards lack large-scale experimentation, thus requiring adaptations in their core to really achieve a high degree of automation when targeting a

Legacy2Cloud logic. In this scope, this paper stresses reverse engineering activity, by showing and illustrating how modeling language can be rationally treated.

In reverse engineering, top-down approaches promote a recovery process that is conducted through architectural knowledge. On the other hand, bottom-up approaches [4] are more pragmatic, since reverse engineering activities lean on source code that is undoubtedly the most rich, self-contained and straightforwardly available material. REMICS focuses on the latter due to the dispersion, or even absence of knowledge. So, model transformations amount to inferring such knowledge from model details and from the expressiveness of their associated modeling languages. Risks occur when the knowledge in models has to move from one language formalism to another. For turning code written in a given programming language into a language-agnostic model, the OMG's task force on modernization puts forward two metamodels: the ASTM [13], which is a metamodel dedicated to syntax trees, and the KDM [12], which is a more global metamodel where the sub-parts deal with a wider spectrum of program-level elements (i.e., user interfaces, data, functions/services, running platform). Unfortunately, KDM and ASTM are not clearly linked to each other. They aim at being complementary but practice shows an unsound, ill-formalized dependency between them. To bridge this gap, this paper discusses, develops and illustrates an intermediary metamodel called SMARTBRIDGE, which allows a roundtrip relationship between KDM and ASTM and hence supports multiple iterations during the reverse engineering activity.

The remainder of the paper is organized as follows: in Section 2 we describe the motivations and the various specifications that gave birth to KDM and ASTM respectively. The SMARTBRIDGE is then detailed in Section 3, including an enumeration of the metaclasses and metarelationships involved in the junction of ASTM and KDM. To make these ideas more concrete, we provide in Section 4 a demonstration of SMARTBRIDGE on a tiny COBOL snippet. Section 5 gives an overview of the related works on model-driven modernization. We conclude this paper in Section 6.

2. Problem of Interest

Recently, MDD standardization has stressed modernization with the special objective of providing new concepts, tools and processes when moving legacy software to renewed applications/information systems running on top of the most up-to-date technologies. The idea behind that is to switch between different technical spaces [10] by underestimating the purely "grammarware" approach in favor of the "modelware" approach. Beyond traditional code-to-code approaches, model-centric approaches are those promoted by MDD in general: (a) conformance to well-defined metamodels, (b) powerful transformation techniques, (c) easier integration of various concerns. Standards in this area also emphasize wider interoperability. In this case, standard-compliant models should be exchangeable between tools involved in the migration chain. These tools process software elements having different shapes. These shapes include the lowest levels (code), as well as the highest: business rules process extraction and interpretation, dealing with the software architecture and components, exhibiting services (business functionalities), etc.

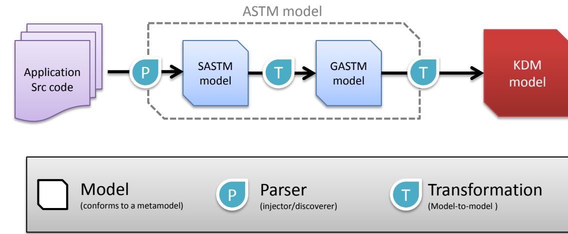

Figure 1. Reverse Engineering process according to the OMG's task force on modernization.

2.1. MDD-based Reverse Engineering

A modernization process encompasses two stages: reverse engineering and forward engineering. The forward engineering stage starts from a Platform-Independent Model (PIM) that serves as the basis for code generation. The reverse engineering stage extracts elements from legacy code and data description, rendering them into a Platform-Specific Model (PSM). KDM is the support candidate for representing PSMs by using ASTM as sub-support for the precise and comprehensive representation of a software system. The creation of PIMs (or technology-neutral models) is favored by the construction of formal mappings between KDM/ASTM on one side and UML (for PIMs) on the other side. More generally, metamodeling fosters the description of discrete steps to show how, at the very beginning, the rough code may be interpreted and analyzed in terms of architectural incidence: operating system adherences, business value discovery strategies, etc.

In this article, KDM is the pivot metamodeling language for representing entire enterprise software systems; including source code of course, but not exclusively. As a common intermediate representation for existing software systems, KDM is a good support for refactoring, the derivation of metrics and the definition of specific viewpoints. Figure 1 depicts the full reverse engineering process promoted by MDD modernization standards.

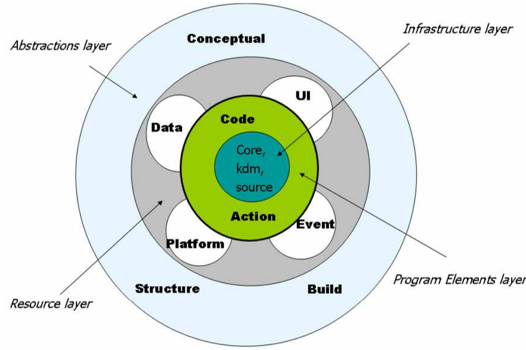

Figure 2. KDM consists of 12 packages arranged into 4 layers

2.2. MDD technologies for modernization

The concomitant use of KDM and ASTM requires a clear understanding of their current capabilities.

2.2.1 Knowledge Discovery Metamodel (KDM)

The architecture of KDM is arranged into four layers, namely the Infrastructure Layer, Program Elements Layer, Runtime Resources Layer and Abstractions Layer (Figure 2). Each layer is dedicated to a particular viewpoint of an application. In this paper we are only interested in the two main layers: the Runtime Resources Layer and the Program Element Layer. These layers allow one to represent the user interfaces, data and code of the legacy application.

The KDM Runtime Resources Layer The Runtime Resource Layer is composed of several packages (`Data`, `UI`, `Event` and `Platform`). However, we will concentrate on the `Data` and `UI` packages herewith. The first one is used to represent the organization of persistent data, especially to describe complex data repositories (e.g., record files, relational schemas, . . .). The second one is used to represent the structure of user interfaces and the dependencies between them in terms of interactions and sequences.

The KDM Program Elements Layer Special attention is paid to the Program Elements Layer, whichis concerned with program-level artifacts. The Program Elements Layer is composed of the `Code` and `Action` packages. The `Code` package represents programming elements as determined by programming languages (data types, procedures, classes, methods, variables, etc.), while the `Action` package describes the low-level behavior elements of applications, including detailed control and data flows resulting from statement sequences. In this scope, KDM is recognized as a way of representing the source code, if only at the execution flow level.

2.2.2 Abstract Syntax Tree Metamodel (ASTM)

As a complement, ASTM has been developed in accordance with the theory of languages to support the representation of source code. In fact, ASTM is composed of the GASTM (Generic Abstract Syntax Tree Metamodel), a standardized language-independent metamodel and the SASTM (Specific Abstract Syntax Tree Metamodel), a user-defined metamodel closely connected with a particular language (Java, COBOL and so on).

Generic Abstract Syntax Tree Metamodel (GASTM) GASTM enables the representation of the code without any language specificity. GASTM contains all of the common concepts of existing languages in the form of metatypes. Parsing some files means instantiating these metatypes along with creating links in order to model semantic dependencies between text pieces. The goal of GASTM is to provide a basis for SASTM in order to later help users to define the domain-specific features of the code to be parsed. The key achievement is to avoid an unintelligible separation (especially in terms of representations) between the 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 below).

Specific Abstract Syntax Tree MetaModel (SASTM) SASTM is constructed through metatypes/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 be dealt 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 or SASTM. The formal interrelation between the two metamodels ensures that models (or their respective instances which represent a given business case) are also consistently linked together based on (fully explicit) comprehensive links.

2.3. Realizing modernization

OMG provides a set of standard specifications for software modernization but fails to provide a guideline for the practitionners. As such, we experimented on the aforesaid technologies while endeavoring to adhere to the – somewhat idealized – process depicted in Figure 1.

2.3.1 Complementarity of MDD technologies

The complementarity of KDM and ASTM resides in the possible code level representations and the different operations that can be applied. On one hand, ASTM permits one to represent a given code source at procedure level, in the form of a syntax-tree whose production has 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 (e.g., data inputs, data outputs, sequences). In fact, a flow level representation provides a direct support for flow analysis and the refactoring strategies thereof. In addition, any reverse engineering process relying on KDM models is reusable, whatever the source technology may be. So, ASTM deals with common parsing issues, while KDM deals with another viewpoint; thus creating the link with other facets like user interfaces, components and so on.

2.3.2 Discretization of the process

The modernization process proposed in this paper is divided into three codified steps:

1. The first step consists in the abstraction of data, code and user interfaces from the legacy material. This is transformed into several technology-specific models. Practically speaking, for each artifact we define its own parser. The parsing outputs are concrete syntax trees (CSTs). Next, these CSTs are transformed into ASTs, each conforming to predefined SASTM metamodels. Starting from chunks of text and ending up as models, this global process is often called ”Injection” in the MDD jargon.

2. This second step consists in the transformation of SASTM models into KDM models, with respect to user interfaces, data and code packages. The goal of this model transformation is to eliminate all technological specificities of the input code model. For that purpose, a reformulation is sometimes required.

3. The third step consists in transforming the code-related and data-related KDM models into UML models, thus generating models in a widely accepted format with their associated graphical notation. There are no technical difficulties except that of choosing between the (existing) concurrent UML-like Java representations that are tolerated by today’s modeling tools.

2.3.3 Current limitations

Our experience has led us to conclude that a modernization project is not a straightforward process, but instead a strongly iterative process, including the re-examination of the models’ parts and their mutual enrichment. We thereby advocate roundtrip capability as to enable information exchange and knowledge propagation within the process, at any level or step. At least two reasons explain this.

First, real modernization requires incorporating additional or derived knowledge into models, either automatically or manually. The most prominent examples are the following:

- Detection of code patterns. The purpose is to recognize sets of object codes that will facilitate refactorings, especially when targetting a completely new architecture.

- Determination of components fate (and the traceability thereof). In accordance with application experts, the purpose is to stamp components that will be migrated and those that will be instead replaced by off-the-shelf components.

- Extraction of business rules. The purpose is to gain a better comprehension of the business logic that underlies a huge amount of lines of code.

- Multi-view modeling. The purpose is to set up the right semantic relationships between the interrelated views of the system that is being reversed (such as user interface, business code and those data structures used for persistence).

Secondly, the strict discretization of the modernizing process envisioned by OMG is not realistic when aiming at providing a modernization CASE tool with a good user-experience level. Indeed, the way a user perceives the computer-aided modernization is an important question, from a tooling viewpoint. Typically, the new knowledge must be impacted to the code model and showed to the user. Thus, some code blocks within a code editor will be highlighted as patterns, while some others will be tagged as “to be replaced”, etc. In other words, the new knowledge derived from the KDM-level must be brought up to the ASTM-level.

3. Proposed research

The observations above stress the necessity to take advantage of the two worlds, while providing roundtrip capability. This means preserving the support for architecture, data, user interfaces (even metrics) that is provided by KDM; as well as achieving the level of details allowed by ASTM. Therefore, we suggest that ASTM and KDM could be interrelated thanks to a bridge that we have dubbed SMARTBRIDGE. This bridge will ensure inter-relationships as well as intra-relationships. The former deal with links between two distinct metamodels: KDM and ASTM. The

latter deal with links between the layers that belong to the KDM metamodel itself.

Building SMARTBRIDGE has led us to focus on three important features:

1. interfacing KDM and ASTM via joint points
2. ensuring navigability between them
3. mapping the different KDM layers: code, data and user interface

3.1. Interfacing

The linkage between the low level code representation allowed by ASTM and the more abstract level allowed by KDM is not natively provided. To overcome this issue, SMARTBRIDGE interposes a number of meta-classes that are showed in table 1.

ASTM	SMARTBRIDGE	KDM
TypeDefinition	EDataType	DataType
DataDefinition	EDataElement	DataElement
TypeDeclaration	EDataType	DataType
AggregateTypeDefinition	EDataType	RecordType
AggregateTypeDeclaration	EDataType	RecordType
FunctionDeclaration	EControlElement	CallableUnit
FunctionDefinition	EControlElement	CallableUnit
VariableDeclaration	EDataElement	StorableUnit
VariableDefinition	EDataElement	StorableUnit
Statement	EActionElement	ActionElement

Table 1. Meta-classes for interfacing ASTM and KDM

Achieving this interfacing calls for an extension point. It turns out that such an extension point was provided in the KDM specification by way of the meta-class `CodeElement` which belongs to `Code` package. Figure 3 shows how SMARTBRIDGE exploits this extension point in order to introduce the required metaclasses to bridge toward ASTM.

Based on the KDM CodeElement meta-class, SMARTBRIDGE defines several meta-classes which are sub-classes of `EElement`. These meta-classes are `ECodeElement`, `EActionElement`, `EDataElement`, `EControlElement` and `EDatatype`. Figure 4 presents an interfacing example between an ASTM `Statement` and its KDM corresponding representation.

3.2. Navigability

We decided to provide a bidirectional navigation capability between KDM and ASTM in order to be able to obtain an ASTM code representation from an abstract code element representation, and conversely. The navigability implemented in SMARTBRIDGE is inspired by the relationship mechanism used in the KDM specification [12]. The KDM code package provides a natural extension point for this relationship mechanism through the `CodeRelationship` meta-class (Fig. 5). SMARTBRIDGE specializes this extension point in two metaclasses: `ERelationship` and `EAggregateRelationship`.

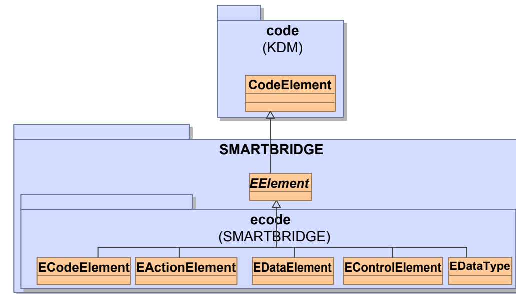

Figure 3. CodeElement extension point

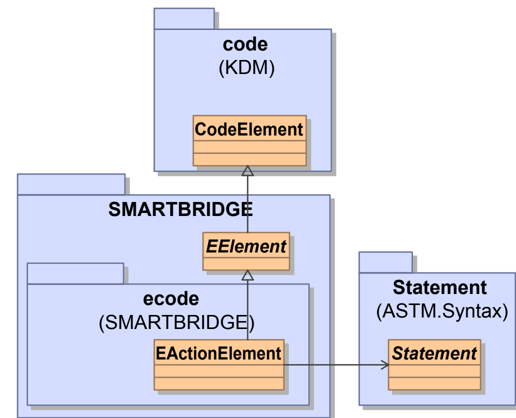

Figure 4. EActionElement meta-class

3.2.1 ERelationship

The `ERelationship` meta-class defines the navigability between KDM meta-classes and the SMARTBRIDGE meta-classes. An `ERelationship` instance is used in order to navigate from a KDM `CodeElement` to a SMARTBRIDGE `EElement` and vice versa. This relationship is exclusively used for a one-to-one relationship, like a function representation (ASTM `FunctionDefinition`), which corresponds exactly to one abstract representation (KDM `CallableUnit`).

3.2.2 EAggregateRelationship

The `EAggregateRelationship` meta-class defines the navigability one-to-many between a KDM element and many SMARTBRIDGE elements. This one-to-many relationship is useful in order to represent KDM abstract elements (like KDM `ActionElement`) using many concrete ASTM ones (like `Statement` metatype).

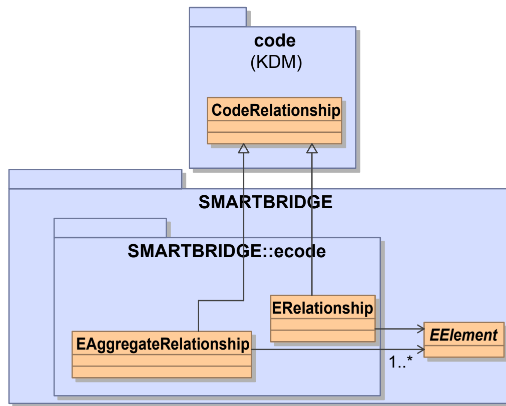

Figure 5. CodeRelationship extension point

3.3. Mapping

The need to provide a mapping between the type definition contained in the legacy source code and the data definition in a database or in the user interface is extremely strong. The KDM meta-model does not provide the concept of specific mapping between the different layers: code, data and user interface. For this reason SMARTBRIDGE introduces this lacking concept through a new package: the `Mapping` package. It defines the link between the source code and the UI from one side and the source code and the data from the other side. The package introduces a new KDM code model called `MappingModel` (c.f. Figure 6). This model contains the existing mapping set representing the data structures, the UI and the source code.

SMARTBRIDGE includes the mapping concept through the abstract meta-class `MappingElement`. Thereby two concrete meta-classes are defined to map the KDM code package element with the KDM data package element `DataMapping` and the KDM ui package element `UIMapping` (Fig. 7).

Thanks to these three important features, i.e. navigability, interfacing and mapping, SMARTBRIDGE fills the gap between the KDM and the ASTM metamodels. This is essential to reach a suitable level of abstraction which enables a better understanding of the legacy source code, and hence an easier modernization activity.

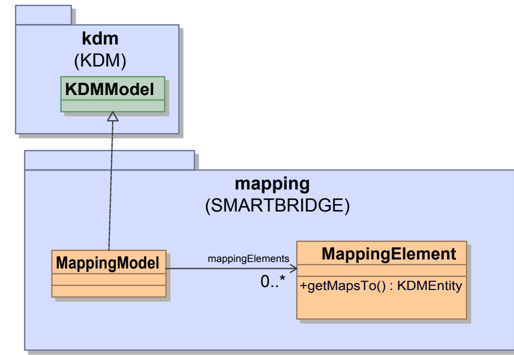

Figure 6. Mapping Model

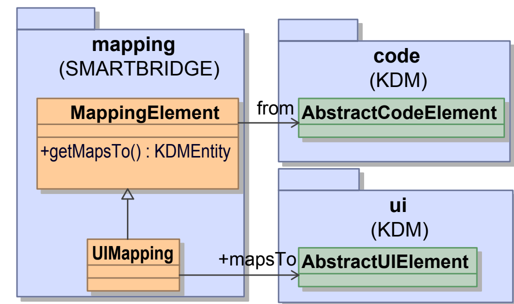

Figure 7. Focus on UI Mapping

4. Working example

This section provides an illustration of SMARTBRIDGE for the modernization of the COBOL legacy code. This illustration especialy focuses on the benefits of interfacing KDM and ASTM. For the sake of simplicity, the following illustrations are based on the tiny COBOL code snippet below:

```
MOVE "sample_error" Error-Message
PERFORM FERROR
```

The modernization process is based on the three following steps:

1. Text to Model transformation (aka. Injection)
2. Interfacing
3. Abstraction refining

4.1. Injection

The first step in the modernization process as discussed in Section 2.3.2, is to obtain an abstract syntax tree conforming to the ASTM metamodel. This step is crucial in

extracting relevant information contained in the code. Thus to build this ASTM model, a COBOL grammar must be used to parsing the legacy source code. The result of this parsing phase is then used to obtain the SASTM (Right part on Fig. 8).

The obtained SASTM model contains specificities closely related to the COBOL programming language like `MOVE`, `PERFORM`, etc. In order to obtain more abstract elements, the transformation of this model into a GASTM one is required (not showed here).

4.2. Interfacing

This second step aims at interfacing between ASTM and KDM by using our SmartBridge metamodel in order to progressively raise the abstraction level and also to iteratively enrich the target KDM model. This interfacing is illustrated in Figure 8.

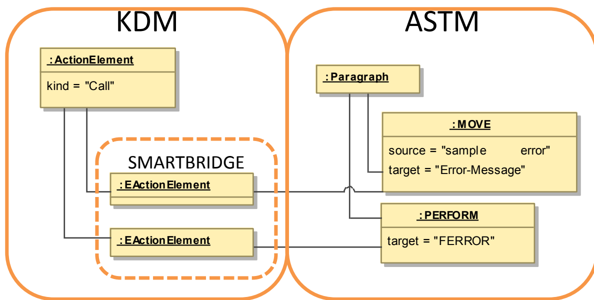

Figure 8. Interfacing example

4.3. Abstraction refining

The third and last step of our modernization process aims at reaching a first abstraction level and at enriching the initial model. In order to accomplish this step, a true source code comprehension is necessary. In fact the `MOVE` instruction (see COBOL code sample) initializes the value `Error-Message`. This field represents the parameter. The KDM representation of the `PERFORM` statement is a CallableUnit call with a parameter value of '*sample error*'. Figure 9 shows the KDM representation obtained using SmartBridge.

5. Related Works

In this section, we study other works that address modernization issues through model-driven technologies. Logically, we pay a special attention to those focusing on the code level of legacy applications. In contrast with KDM-compliant approaches, approaches relying on proprietary metamodels, tailored for particular usages do exist.

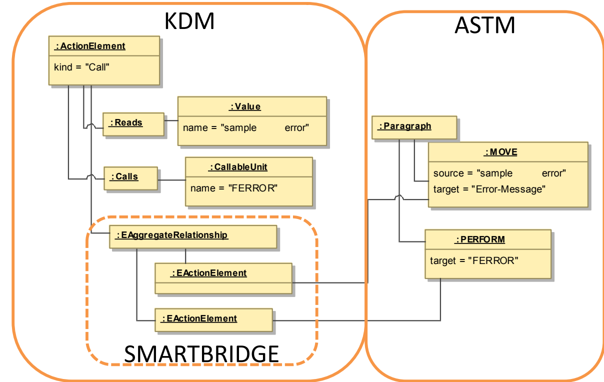

Figure 9. Abstraction refining example

5.1. KDM-uncompliant

Reus *et al.* in [15] propose a MDA process for software migration where they parse the text of the original system and build a model of the abstract syntax tree. This model is then transformed into an intermediate langage dubbed GenericAST that can be translated into UML.

In [6], the authors summarize the use and impact of the TGraph technology in Reverse Engineering. TGraphs [5] are directed graphs whose vertices and edges are typed, attributed, and ordered. In fact, representing source code as a typed graph can be rephrased as representing code as a model conforming to a metamodel. From this point of view, metamodels used by parsers are designed from scratch.

Izquierdo and Molina developed the Gra2MoL approach [3], where a model extraction process is considered as a grammar-to-model transformation, so mappings between grammar elements and metamodel elements are explicitly specified. Beyond the technical aspects of the proposed transformation language, one may notice that the target metamodels are user-defined.

Fleurey *et al.* describe in [9] a model-driven migration process in an industrial context. For that purpose, a tool suite for model manipulation is used as a basis for automating the migration. The reverse engineering step moves from a code model (output of the parsing) to a PIM, which is implemented by model transformations from a legacy language meta-model (e.g., COBOL) to a pivot metamodel. The pivot metamodel is called ANT and contains packages to represent data structures, actions, UIs and application navigation.

5.2. KDM-compliant

In [14], Perez-Castillo *et al.* propose a technique that recovers code-to-data links in legacy systems based on relational databases and enable one to represent and manage these linkages throughout the entire reengineering pro-

cess. The proposal follows the ADM approach by leveraging KDM, especially the code package of the Program Elements Layer where SQL sentences have been modeled through a KDM extension. In this case, it is not an actual metamodel, it is a profile one instead.

MoDisco (Model Discovery) [1] is the model extraction framework part of the Eclipse GMT project (www.eclipse.org/gmt). This framework is currently under development and provides a model managing infrastructure dedicated to the implementation of dedicated parsers ("discoverers" in MoDisco terminology). A KDM-based metamodel, a metamodel extension mechanism and a methodology for designing such extensions are also planned.

6. Conclusion

ASTM and KDM complement each other in modeling software systems' syntax and semantics. In this article, we propose to fill the gap between the two by introducing SMARTBRIDGE in order to remain in the scope of ADM and hence ensure the interoperability of the outputs of MDD reverse engineering activities.

Gluing ASTM and KDM aims at overcoming the main flaw of a strict discretization of the modernization process, thus enabling roundtrip engineering. It also balances out the purely low-level representation of the legacy material supported by ASTM and the higher abstraction level supported by KDM. Hence, SMARTBRIDGE has been implemented within BLUAGE® (www.bluage.com) and has been proven to better represent code, while maintaining good architectural representation; rather than using KDM and ASTM separately. As such, we have demonstrated – in a tooling purpose – that SMARTBRIDGE supplies a practical answer to the traceability from end-to-end issue, along with knowledge propagation at every step.

We are currently improving SMARTBRIDGE with additional features. Indeed, similarly to the glue ASTM-KDM, we plan to fill the gap with further OMG ADM metamodels like SPAP (Software Patterns Analysis Package), SMM (Software Metrics Metamodel), etc. These new relationships will make it possible to handle the different aspects of a legacy system as a cohesive whole.

Acknowledgements

This work has been funded by the European Commission through the REMICS project (www.remics.eu), contract number 257793, within the 7th Framework Programme.

References

[1] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. Modisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM.

[2] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7:13–17, January 1990.

[3] J. Cnovas Izquierdo and J. Molina. A domain specific language for extracting models in software modernization. In R. Paige, A. Hartman, and A. Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 82–97. Springer Berlin / Heidelberg, 2009.

[4] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. Softw. Eng.*, 35:573–591, July 2009.

[5] J. Ebert and A. Franzke. A declarative approach to graph based modeling. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG '94, pages 38–50, London, UK, 1995. Springer-Verlag.

[6] J. Ebert, V. Riediger, and A. Winter. Graph technology in reverse engineering: The tgraph approach. In *Workshop Software Reengineering*, pages 67–81, 2008.

[7] L. Favre. *Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution*. Premier Reference Source. Igi Global, 2010.

[8] F.Barbier, G.Deltombe, O.Parisy, and K.Youbi. Model driven reverse engineering: Increasing legacy technology independence. In *The 4th India Software Engineering Conference*, Thiruvanantpuram, India, February 2011. CSI ed.

[9] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel. Model-driven engineering for software migration in a large industrial context. In *MoDELS*, pages 482–497, 2007.

[10] I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA 2002 Federated Conferences, Industrial track*, 2002.

[11] A. S. F. B. Mohagheghi Parastoo, Berre Arne Jrgen and G. Benguria. Reuse and migration of legacy systems to interoperable cloud services - the remics project. In *Proceedings of Mda4ServiceCloud'10 at the Sixth European Conference on Modelling Foundations and Applications*, ECMFA '10, June 2010.

[12] OMG). Knowledge discovery metamodel - version 1.3. http://www.omg.org/spec/KDM/1.3, 2011.

[13] OMG. Syntax tree metamodel - version 1.0. http://www.omg.org/spec/ASTM/1.0, 2011.

[14] R. Perez-Castillo, I. G.-R. de Guzman, O. Avila-Garcia, and M. Piattini. On the use of adm to contextualize data on legacy source code for software modernization. *Reverse Engineering, Working Conference on*, 0:128–132, 2009.

[15] T. Reus, H. Geers, and A. van Deursen. Harvesting software systems for mda-based reengineering. In *ECMDA-FA*, pages 213–225, 2006.

[16] W. Ulrich. A status on omg architecture-driven modernization task force. In *Proceedings EDOC Workshop on Model-Driven Evolution of Legacy Systems*, Monterey, California, USA, 2004. IEEE Computer Society.