Model Driven Reverse Engineering: Increasing Legacy Technology Independence

Franck Barbier**, Gaëtan Deltombe**, Olivier Parisy*, Kamal Youbi*
*Netfective Technology, France
+University of Pau, France
franck.barbier@univ-pau.fr, {g.deltombe, o.parisy, k.youbi}@netfective.com

Abstract

In this paper, we describe a model driven reverse engineering method supported by a tool named BLU AGE®. This method initially suffered from a lack of genericity due to its development and tuning within the context of two large-scale experimentations linked to a given legacy technology. We concisely expose this method: its strengths especially resulting from metamodeling and model transformation, two core notions of Model Driven Engineering (MDE). We explain the forthcoming actions to generalize it by using the Knowledge Discovery Metamodel (KDM) standard in the ReMiCS European project especially.

1. Introduction

To meet changing market needs, to increase productivity and quality, companies endow themselves with a regularly growing set of corporate applications. When this set of software starts getting old (typically tens of years), one notices a lack of agility, even strong problems to evolve those applications written in “legacy” technologies (as an example, in COBOL). Evolution often stumbling over a lost of know-how and knowledge: over years, both legacy technologies and business rules engraved in applications are no longer known. Concomitantly, complexity of newest technologies is important along with poor interoperability to connect old information systems with Internet for example.

Paradoxically, the business value of old applications is higher and higher; their role is strategic even critical; their potential impact on business grows. The core activity of companies thus depends upon information systems in an excessive way. The central place occupied by these applications in the business processes of corporations explains the need for modernization of this legacy capital, i.e., the migration to more up-to-date modern supports. This modernization aims at making the migrated applications easier to maintain by people not (or barely) knowledgeable of the original legacy technology. This migration approach encompasses the idea of reverse engineering [1]: the process of studying an existing system so that its inner workings can be discovered and reformulated in an efficient way.

In this paper, we show that MDE provides a practical answer to reverse engineering [2]. MDE provides notions (and powerful tools) to represent complex systems as models (UML models in general), hence offering more malleable, transformable software artifacts. Legacy applications and pieces are viewed as instances of metamodels through several (more or less abstract) shapes depending upon the current stage of the reverse engineering process. Progresses when moving software from its legacy status to modern environments are realized by means of discrete steps, i.e., model transformations expressed as mappings between metamodels.

The key issue with model driven reverse engineering is to produce models as outputs of legacy code processing (a.k.a. “essential reverse engineering” in the rest of the paper). These models are in essence free from technological constraints (PIMs standing for Platform-Independent Models) to rub out, as much as possible, legacy technology constraints. Next, PIMs themselves serve as inputs in application generation processes. These processes are today’s often called SOAization processes because renewed applications
move to Service-Oriented Architectures (SOAs) including cloud computing.

In this paper, we discuss the fact that the proposed reverse engineering method cannot be fully automated. Reverse engineers must then drive the reverse engineering process through appropriate tool assistance implying “interaction” with the software material to be reversed. So, being free from the legacy technology is a great challenge when one wants, in particular, to go beyond code-to-code approaches. Despite the abstract nature of models, a detailed representation of the whole code is required. In this scope, the Xtext Eclipse component allows the definition of specific grammars to later describe source code as direct formal KDM models.

Two industrial experimentations evoked in this paper illustrate the use of KDM along with the necessity to extend this standard. This approach aims at first generalizing the proposed reverse method while keeping valuable results like, for instance, code pattern recognition to create software services in modernized applications. The second point is the adaptation of the BLU AGE® Reverse module [3], an Eclipse IDE plugin. New challenges are to deal with distinct unrelated legacy technologies in ReMiCS [4-5]: multiple COBOL dialects but also, for instance, PL/SQL, an Oracle dialect.

The paper exposes the method developed and used to successfully carry out two large-scale case studies based on the same legacy technology: Visual Age Pacbase (see Section 2). The paper later explains (Section 3) the tasks to be performed to generalize this method by extending KDM. Indeed, the method is based on a seamless chain of model transformations that leads to UML models at the end of the chain. The migration phase requires UML models as inputs of an application generation process. While KDM is the metamodeling support for reverse, final results must be understandable by stakeholders and interpretable in general-purpose modeling tools, out of the perimeter of reverse engineering especially. That is the reason why the very last transformations move the predigested KDM models to UML PIMs (Section 4). Section 5 sketches forthcoming important challenges in a conclusion.

2. Reverse Engineering Context and Method

Les Mousquetaires, one of the biggest players in the retail sector in Europe, and Wellmark, a US health insurance company, have their information systems constructed on the top of Visual Age Pacbase (VAP), an IBM fourth-generation language (a superset of COBOL in fact) itself surrounded by an integrated maintenance environment. This last tool allows a better application structuring compared to COBOL programming in a “flat” way. Data and code are organized through a repository made up of tables. These tables may be easily queried through Pacbase Access Facility (PAF). PAF serves as a primary extractor to start up the reverse engineering process.

Parts of the information systems of these two companies have been transformed into rich UML models with the help of the BLU AGE® Reverse module. Next, the BLU AGE® Design&Generate module offers abilities to generate renewed applications from these models. BLU AGE® Design&Generate is nothing else than a UML profile, a set of markers to annotate technology-free models. These models are next injected into a generator, the second component of the BLU AGE® Design&Generate module, to produce Java EE applications for example. These applications at least offer, in number and in quality, equivalent functions compared to the legacy system.

In each of these industrial experimentations, representative batch and transaction processing programs (those associated with old-fashioned character screens) were selected with great care to be reversed. This gave birth to a reverse engineering method implemented in the BLU AGE® Reverse module [6]. This method mainly consists in the availability of ad hoc VAP-specific metamodels to represent VAP applications (at different stages of the discovery and consolidation processes) as metamodel instances. The very rough nature of the legacy material requires several transformation stages (Figure 1).

![Figure 1. Initial reverse steps](image)

In this approach, a textual Domain-Specific Language (DSL) is constructed by means of Xtext. One DSL (a grammar in fact) is required per dialect. For instance, VAP COBOL and Micro Focus COBOL lead to two distinct DSLs. Parsing the rough code allows the production of KDM models. KDM currently applies for being more reverse-centric than UML. Indeed, KDM enables to simultaneously representing an application at different levels of granularity (Figure 2): statements, data records, user interfaces, execution features like system calls… KDM is itself a metamodel (TrueFlow, FalseFlow, ActionElement, CallableUnit… are KDM metatypes). BLU AGE® Reverse makes possible the formalization of any reverse activity as a model transformation using the ATL [7] model.
transformation language. Once KDM models are available, one moves technology-dependent models to technology-free models. In fact, a KDM metamodel adaptation is required to represent any legacy technology in general (top of Figure 2).

![Figure 2. Revisited reverse process with KDM in the context of the VAP COBOL dialect](image)

From now, in both industrial cases, VAP played the role of the single legacy technology. In contrast, the ReMiCS project aims at elaborating on different technologies. A given application may run on the top of several distinct legacy technologies: for instance, an application with one backoffice component in Micro Focus COBOL with another based on .NET/C#. Moreover, several concern-oriented models must represent the legacy system under different perspectives: code, data dictionary, architecture, business rules... with tight interrelation across models. A key challenge is to capitalize the current method along with the implementation of two new use cases in ReMiCS: an “invoice reminders” module to be added as service to a more global “Software as a Service” accounting application on one hand. On another hand, we stress a trip booking application used by tour operators, hotels and other key players in this business area.

2.1. Extraction

The extraction step aims at acquiring information from a source in an exhaustive way. This step produces a rough model (Figures 1 & 2) created from a textual definition (flat files) of the legacy application. VAP offers a smart extractor for that (PAF). However, ordinary legacy systems are poorly structured in general. Even though code extraction (under the form of trees especially) is not a tough task to be performed, extraction may benefit from the early highlighting of primary dependencies between legacy artifacts.

To generalize, a first KDM metamodel is required for, as much as possible, organizing the extracted software assets to achieve a good separation of concerns. This systematically involves manual intervention to give contextual semantics to these assets through available markers/annotations offered by BLU AGE®. System calls (CICS calls in COBOL for example) for instance must be localized and marked with great care to avoid their useless migration in the renewed application that, in essence, uses different system facilities.

2.2. Interpretation

This step aims at transforming the model generated during the extraction step to models describing the application to reformat in an object-oriented way: a dictionary of business objects, other meaningful objects as well (user interfaces...) at a higher abstraction level. In fact, we have to clean up models stained by technology properties to obtain Platform-Independent Models (PIMs) to move to non-legacy platforms. Several micro-steps are required but we only list some of them in this paper.

The interpretation phase may be conducted in two non-exclusive ways. The first one called “essential reverse engineering” (Section 2.2.1) is based on a one-to-one mapping between the original code and its representing model. However, the resulting model is poor in terms of “semantics”. For instance, many legacy technologies do not rely on a strong modularization of the code. There are few separations of concerns in the code. Code may be decorated with configuration data, internally with C/C++ macros, Java annotations or, externally with XML files, etc.

The second manner named modernization calls for a greater intervention of software engineers to introduce semantic tags in the code. The key point is that this process will be supported in the future by a knowledge base in charge of capitalizing knowledge about the legacy technology. We expect more or less automatic learning here. This means that the KDM model of the legacy technology (top of Figure 2) is not built one and for all. Instead, it is incrementally enriched during reverse.

2.2.1. Essential Reverse Engineering. Essential reverse engineering corresponds to the production of a model from source code in a bijective way. For instance, each VAP COBOL program statement leads to a KDM model element, namely an instance of ActionElement.

This approach is interesting in a quick code migration context. Its key benefit is traceability. However, it is severely limited when one wants to rationalize even evolve the code (one may take advantage of enhancing functionality for instance), and not only perform a one-shot platform migration. Of course, essential reverse
engineering can be used in well-delimited pieces of code for the modernization phase if no significant reconsiderations are required.

2.2.2. Modernization. Modernization is about understanding and deeply restructuring the legacy application code with semantic effects. Currently, the modernization phase is deliberately not automated since no “intelligent technique” exists to interpret the code without upstream analysis. Code constructs like blocks, subprograms, types… are restructured, simplified, improved, classified, even suppressed. To help engineers, a set of consistent related views are offered within Eclipse. Those views are ruled by an editor dedicated to the management of the legacy code. This global editor is itself supplemented by an annotation editor.

2.2.3. Eclipse Views. Each Eclipse view has a specific goal: supplying the user with information about an element with respect to the context in which it appears. Elements can be represented in different views under different forms at the same time: as a legacy artifact versus its modernized form in a model. One key objective is indeed to obtain an explicit mapping, to let the possibility to explain why something appears in a model through a coherent chain of traced changes.

The annotation editor (Figure 3) is used to add accurate indications to the code (right hand side), from neutral comments (blue) to meaningful annotations that greatly influence the translation process when moving annotated elements to modernized objects.

2.2.4. Code Pattern Recognition. Code pattern characterization is an important part of the modernization process since it aims at automatically transforming all instances of a given parameterized pattern (discovered in the legacy code) into a predefined model element.

This supposes the definition of code templates (bottom of Figure 4) with parameters (between <> and their matching with code pieces (top of Figure 4). Discovering and next marking patterns through appropriate (homemade) annotations like Skipped, Pattern, Modernized Construct… forces the reverse engine to identify, and later factorize, code copies. At this time, pattern recognition is independent of a given legacy code dialect. Instead, an extended grammar has been defined to support this mechanism.

![Figure 3. Annotation editor](image)

![Figure 4. Pattern recognition in COBOL](image)

3. KDM Extensions

The solution we are currently investigating and developing relies on an extension of KDM along with an early implementation\(^1\) of the forthcoming Abstract Syntax Tree Metamodel (ASTM) standard.

Technically, KDM includes Micro-KDM for low-level modeling (code) while other KDM packages emphasize architectures, business rules… modeling. Moreover, the complementarity between KDM and ASTM is not clear even confusing. Indeed, gateways are planned instead of creating a good integration from scratch.

The code package of KDM (a.k.a. Micro-KDM) allows the representation of a statement of the legacy language at both a micro and macro levels. There are several micro objects attached to a given macro object, *i.e.*, an instance of the ActionElement metatype. Micro objects express the precise meaning of a statement. For an assignment, a left-operand, a right-operand and the assignment symbol itself. However, the semantics is implicit and often linked to the legacy language. More generally, a semantic construction has the same effect at runtime with possible different forms in different legacy dialects: `MOVE` in COBOL, setter in Java, etc.

---

\(^1\) Despite the availability of a beta specification, no implementation exists.
The proposed KDM extension is to first add metatypes systematically describing KDM micro-actions. For instance, the Assign metatype is introduced as a subtype of ActionElement contrary to the current situation: ActionElement is instantiated with a “kind” attribute having the “Assign” value. Next, metarelationships to existing and new metatypes are also incorporated into the proposed KDM extension. Metarelationships may be endowed with OCL constraints to improve the meaning of KDM metatypes. In other words, to understand the deep sense of a KDM metatype, a priority is the analysis of its relationships with other KDM metatypes and how these relationships behave (OCL constraints) for instances of these metatypes, i.e., for models representing a given legacy system. As an illustration, the Assign metatype which inherits from ActionElement has to be connected with a Reads, Writes and Value metatypes. Beyond COBOL, such a metamodeling pattern may be used to reverse a Java setter or any equivalent construction in PL/SQL for instance.

The general rule is: any language construct must be representable by a KDM metatype along with an instantiation of the metarelationships of this metatype (values/collections of other metatype objects governed, possibly, by OCL constraints).

4. Migration

In the migration phase, patterns are also used to anticipate the structuring of UML models embodying the modernized application. Patterns can be manipulated as parameterized UML models (Figure 5) that aim at being instantiated at generation time for each code piece (Figure 6) conforming to a pattern.

![Figure 5. Eclipse UML model (template) describing an “empty” modernized application](image)

5. Challenges and Conclusion

Any legacy application includes recurrent concerns like presentation, security, persistence... These concerns have non-homogenous shapes in legacy systems. Considering persistence for example, this imposes the isolation of a persistence layer to become, in a systematic way, a canonical class diagram in UML to be used as input of a straight application generation process.

Regarding KDM, this leads to envisaging a standalone persistence metamodel to specifically take care of this concern. The cohabitation of concern-dependent metamodels is based on model weaving. While the KDM metamodel interrelation must maintain the consistency of the represented application, the difficulty remains in the management of different flows to populate the various KDM (parameterized) metamodels. Some of them can be populated from scratch when no concern exists in the legacy code: for example, the absence of security features while the renewed application calls for such features.

More generally, we show in this paper that extensions of KDM are necessary along with the formal characterization of the complementarity between KDM and ASTM. This paper demonstrates some advances on that topic and the need for implementing an efficient version of these standards.

References


**Acknowledgment**

This work has partly been funded by the European Commission through the REMICS project ([http://remics.eu](http://remics.eu)), contract number 257793, within the 7th Framework Program.