Recovering Sequence Diagrams from Object-oriented Code
An ADM Approach

Liliana Martinez[1], Claudia Pereira[1] and Liliana Favre[1,2]
[1]*Universidad Nacional del Centro de la Provincia de Buenos Aires, Paraje Arroyo Seco, B7000, Tandil, Argentina*
[2]*Comisión de Investigaciones Científicas de la Provincia de Buenos Aires, La Plata, Argentina*
{lmartine, cpereira, lfavre}@exa.unicen.edu.ar

Keywords: Architecture-Driven Modernization, Reverse Engineering, Model Driven Architecture, Knowledge Discovery Metamodel.

Abstract: Software modernization is a current research area in the software industry intended to transform an existing software system to a new one satisfying new demands. The initiative Architecture-Driven Modernization (ADM) helps software developers in tackling reverse engineering, software evolution and, software modernization in general. To support modernization problems, the ADM Task Force has defined a set of metamodels such as KDM (Knowledge Discovery Metamodel), being the Eclipse-MDT MoDisco project the official support for software modernization. We propose the application of ADM principles to provide relevant model-based views on legacy systems. We describe a framework to reverse engineering models from object-oriented code. In this context, we show how to recover UML sequence diagrams from Java code. We validate our approach by using ADM standards and MoDisco platform. Our research can be considered a contribution to the MoDisco community; MoDisco does not support reverse engineering of sequence diagrams and, on the other hand, the MoDisco KDM Discover was used and enriched to obtain the required information for recovering interaction diagrams.

1 INTRODUCTION

Nowadays, almost all companies are facing the problematic of having to modernize or replace their legacy software systems. These old systems have involved the investment of money, time and other resources through the ages. Many of them are still business-critical and there is a high risk in replacing them. Software modernization refers to the transformation from an existing software system to a new one that satisfies new requirements. Modernization is related to different processes such as migration, software refactoring, architecture restructuring, and mainly reverse engineering.

OMG is involved in the definition of standards to modernize information systems. In this context, a new approach known as Architecture-Driven Modernization (ADM) (ADM, 2014) has emerged as an evolution of MDA and its standards to support the modernization of systems (MDA, 2014). MDA is the particular OMG vision of Model Driven Development (MDD) being its essence the Meta Object Facility (MOF, 2011). The OMG ADM Task Force (ADMTF) has defined a set of metamodels aligned with MOF that allow describing various aspects of the modernization problems. Metamodels such as Knowledge Discovery Metamodel (KDM) and Abstract Syntax Tree Metamodel (ASTM) aim at improving the process of understanding and evolving software applications and enabling architecture-driven reverse engineering (KDM, 2011); (ASTM, 2011). The Eclipse-MDT MoDisco open source project is considered by ADMTF as the reference provider for implementations of several of its standards (MoDisco, 2014).

Reverse engineering techniques allow supporting an integral part of the software modernization. Reverse engineering involves (re)discovering the functional, structural and behavioral semantics of a given artifact to document, maintain, improve or migrate them. To support reverse engineering, we propose an adaptation of traditional software engineering techniques to the ADM context. We describe a model driven reverse engineering framework to reverse engineering platform independent models, expressed as UML diagrams (UML, 2011), from object-oriented code. We propose the use of OMG standards and the MoDisco

platform to validate our approach. Currently, MoDisco can only recover UML class diagrams from Java code. In a previous work, we show how to reverse engineering use case diagrams in the ADM context (Martinez, Favre and Pereira, 2013). In this paper, we extend the proposal by means of reverse engineering sequence diagrams from Java code. To recover the KDM model, the MoDisco KDM Discoverer was enriched to obtain the required information to recover the sequence diagram due to the transformation provided by MoDisco is not fully specified, there are elements in the Java model that were not fully transformed into their corresponding KDM elements. Then, we implemented a model-to-model transformation to recover sequence diagrams from the KDM model. Thus, our research may be considered a contribution to MoDisco community; sequence diagrams reverse engineering is not supported by MoDisco and, on the other hand, MoDisco KDM Discoverer was enriched to obtain the required information for recovering sequence diagrams and other interaction diagrams.

This paper is organized as follows. Section 2 presents OMG standards, tools and work related to software modernization, particularly reverse engineering. Section 3 describes a framework for architecture-driven reverse engineering. Section 4 presents a study case that shows how to reverse engineering sequence diagrams from Java code. Section 5 analyses the obtained results. Finally, Section 6 presents conclusions and future work.

2 BACKGROUND

Software industry constantly evolves to satisfy new demands. Nowadays, there is an increased demand for software migration as well as modernization of legacy systems that are still business-critical to extend their useful lifetime. The success of system modernization depends on the existence of technical frameworks for information integration and tool interoperability. In this section, we describe OMG standards for modernization. Next, we discuss about languages for model transformation. Finally, we present work related to software modernization, particularly reverse engineering.

2.1 Standards for Modernization

The purpose of standardization is to achieve well-defined interfaces and formats for interchange of information about software models to facilitate interoperability between the software modernization tools and services of the adherents of the standard. This will enable a new generation of solutions to benefit the whole industry and encourage collaboration among complementary vendors.

ADMTF is developing a set of standards of which we are interested in KDM and ASTM. KDM is the foundation for software modernization. KDM represents entire enterprise software systems, not just code. ASTM is a specification for modeling elements to express abstract syntax trees (AST). KDM and ASTM are two complementary modeling specifications. KDM establishes a specification that allows representing semantic information about a software system, whereas ASTM establishes a specification for representing the source code syntax by means of AST. ASTM acts as the lowest level foundation for modeling software within the OMG ecosystem of standards, whereas KDM serves as a gateway to the higher-level OMG models.

2.2 Model Transformation Languages

Query/View/Transformation is the OMG standard language to express transformations on MOF models (QVT, 2011). CASE tools support QVT or at least, any of the QVT languages. The MMT (Model-to-Model Transformation) Eclipse project is a subproject of the top-level Eclipse Modeling Project that provides a framework for model-to-model transformation languages. Transformations are executed by transformation engines plugged into the Eclipse Modeling infrastructure. ATL is a model transformation language and a toolkit that provides ways to produce a set of target models from a set of source models (ATL, 2014). To date, ATL is the most used transformation language due to his maturity degree. It is worth considering that QVT declarative is in its “incubation” phase and only provides editing capabilities.

2.3 Modernization and MoDisco

With the emergence of ADM, new tools need to be developed. To be ADM compliant, these tools should provide features such as support for modeling, interoperability and standardization, automated transformations for both forward and reverse engineering, access to the definition of these transformations and, support for traceability.

Today, the most complete technology that supports ADM is MoDisco which provides a generic and extensible framework to facilitate the development of tools to extract models from legacy systems and use them on use cases of modernization.

As an Eclipse component, MoDisco can be integrated with plug-ins or technologies available in the Eclipse environment. The MoDisco project is working in collaboration with ADMTF. To facilitate reuse of components between several modernization solutions, MoDisco is organized in three layers. The Infrastructure layer contains generic components independent from any legacy technology such as EMF implementations of ASTM, KDM, the KDM Source discoverer, and the KDM to UML converter. The Technology layer contains component dedicated to one legacy technology such as metamodels for the Java language, Java AST and Java Discoverer. The Use-cases layer contains components providing a solution for a specific modernization use-case.

2.4 Related Work

Many works have contributed to reverse engineering object-oriented code. Tonella and Potrich (2005) provide a relevant overview of techniques that have been recently investigated and applied in the field of reverse engineering of object-oriented code. Authors describe the algorithms involved in the recovery of UML diagrams from code. Our proposal can be considered as a formalization of the recovery processes described at Tonella and Potrich (2005) in terms of standards involved in ADM.

Among the works related to MDD-based reverse engineering but not in the ADM context, it is worth mentioning (Izquierdo and Molina, 2009a) and (Deissenboeck and Ratiu, 2006).

With the emergence of ADM, new approaches and tools are being developed. Martinez, Favre and Pereira (2013) describe the state of the art in the model-driven modernization area and discuss about existing tools and future trends. A process to extract models that conform to KDM is presented in (Cánovas Izquierdo and García Molina, 2009b). This approach does not recovery UML models. Barbier et al., (2011) describe a model driven reverse engineering method and illustrate it with two COBOL legacy systems. Authors explain the future actions to generalize it by using KDM.

Several tools support the recovering of sequence diagrams from object-oriented code. Most of them are not based on the principles of MDA and ADM, recent examples are Visual Paradigm (Visual Paradigm, 2014), Java Call Tracer (Java Call Tracer, 2014) and RevEng (Tonella and Potrich, 2005). Blu Age follows the principles of MDA and ADM through Eclipse (Blu Age Reverse Modeling, 2014). This tool has been targeting the modernization of COBOL in particular, to facilitate the transferring of legacy code towards object-oriented technologies of the JEE or .Net type. Our approach has been targeting with a different aim, the modernization of object-oriented code to facilitate the adaptation of legacy applications to mobile platforms.

3 A FRAMEWORK FOR ADM

Three main steps in software modernization: Model Discovery, Model Understanding and Model (Re) Generation are distinguished in (Brambilla et al., 2012). The first phase “discovers” a set of initial models that represent the legacy system at the same abstraction level. The second phase employs query and transformation techniques that built models in a higher-level of abstraction, which are the source models in the Model (Re) Generation phase. The present work describes a framework for architecture-driven reverse engineering (ADRE) to recover models from object-oriented code that focus on the first two above-mentioned steps of the modernization (Figure 1). In the MDD context, the reverse engineering process extracts elements from existing systems and represents them into Platform Specific Models (PSMs), and subsequently Platform Independent Models (PIMs) are obtained from the PSMs. In the ADM context, KDM is the support for representing PSMs by using AST as intermediate representation of a software system. In particular, we describe how to recover models that represent an abstract view of existing systems from its code in the ADM context.

The model level includes code models (Implementation Specific Model - ISM), KDM models (PSM) and UML models (PIM). The last ones provide a uniform representation of the system in the ADM context and include class, use case, activity, interaction, and state diagrams.

The metamodel level includes metamodels defined via MOF that are the foundation to describe the transformations at model level. The metamodel level includes ASTM that describes AST models, KDM that describes families of PSM models and the UML metamodel that describes families of PIMs.

The models at PIM level are built applying successive transformations from source code. For each transformation, source and target metamodels are specified.

The reverse engineering process at metamodel level consists of two major steps:

1. Model Discovery: a code model is obtained by applying a text-to-model (T2M) transformation from source code; it is transformed into an

abstract syntax tree that conforms to ASTM.

2. Model Understanding: the aim is to raise the abstraction levels generating UML models by using model-to-model (M2M) transformations implemented in ATL. This step involves two successive transformations:

2.1. M2M transformation to discover KDM models from code model.

2.2. M2M transformation to discover UML models from KDM models.

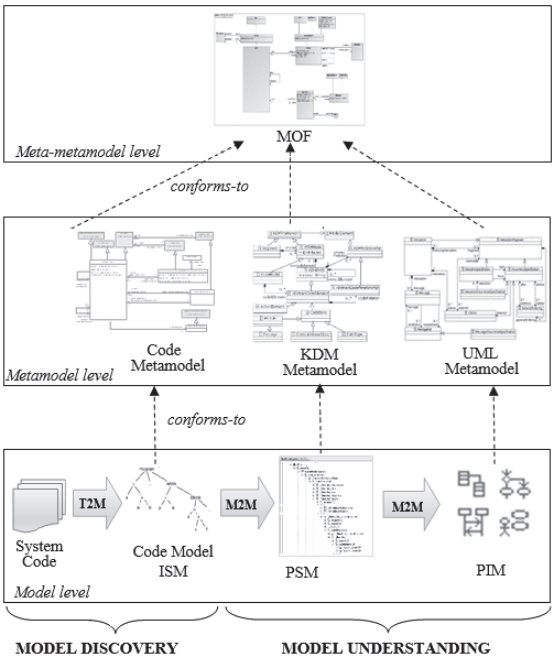

Figure 1: Framework for ADRE.

4 REVERSE ENGINEERING OF SEQUENCE DIAGRAMS

Interaction diagrams are not only important for modeling the expected behavior of a system during forward engineering but also for understanding the system behavior during reverse engineering. In this section, we exemplify a reverse engineering process at metamodel level to recover sequence diagrams from code by using the same case study used in Tonella and Potrich (2005), the Java program *eLib* that support the main library functions. The Java code is partially shown in Figure 2. It supposes an archive of documents of different categories (books, journals and technical reports). Each document is uniquely identified and library users can request document for loan. To borrow a document, a user must be identified by the librarian. While books are available for loan to any user, journals can be borrowed only by internal users, and technical reports can be consulted but not borrowed.

```
class Library {
  Map documents... Map users...Collection loans...
  public boolean addUser(User user) {...}
  private void addLoan(Loan loan) {...}
  public boolean borrowDocument(User user,Document doc)
   {if (user.numberOfLoans()<MAX_NUMBER_OF_LOANS&&
       doc.isAvailable()&&doc.authorizedLoan(user)){
          Loan loan = new Loan(user, doc);
          addLoan(loan); return true;}
    return false; }                                    ...}
class Document {
  int documentCode; Loan loan = null;...
  public boolean isAvailable(){return loan==null;}
 public boolean authorizedLoan(User u){return true;}…}
class Book extends Document {...}
class Journal extends Document {  ...
  public boolean authorizedLoan(User user) {
    return user.authorizedUser();}                     ...}
class TechnicalReport extends Document{...
  public boolean authorizedLoan(User user){...}      ...}
class User {  int userCode; Collection loans ...
  public boolean authorizedUser(){return false;}
  public void addLoan(Loan loan){loans.add(loan);} ...}
class InternalUser extends User { ...
   public boolean authorizedUser() {return true;}  ...}
class Loan {  User user; Document document;
  public Loan(User usr, Document doc) {...}          ...}
```

Figure 2: *eLib* program.

4.1 Model Discovery

The first step of the reverse engineering process at metamodel level consists in discovering models from the existing system. The *eLib* program is written in Java language, thus, we use the JavaAST Discoverer provided by MoDisco to obtain its corresponding AST model. This discoverer creates Java models from Java source code contained in a Java project. This transformation was used as it is provided by MoDisco, with no modification.

4.2 Model Understanding

The second step of the reverse engineering process consists in the transformation of the software model into UML models by two successive transformations described in the next subsections.

4.2.1 Recovering KDM Model

The KDM models are instances of the KDM metamodel that is partially shown in Figure 3. The main metaclasses are *Segment*, *KDMModel*, *KDMEntity* and *KDMRelationship*. Segment is a

container for information about an existing software system. A segment includes *KDMModel* instances representing one architectural view of the system. A *KDMModel* instance owns entities that are named elements that represent an artifact of existing software systems such as packages and classes. A KDM entity owns elements and relationships. *KDMRelationship* element is an abstraction that specifies relationships between entities. Each instance of *KDMRelationship,* such as *Calls* has exactly one target and exactly one origin.

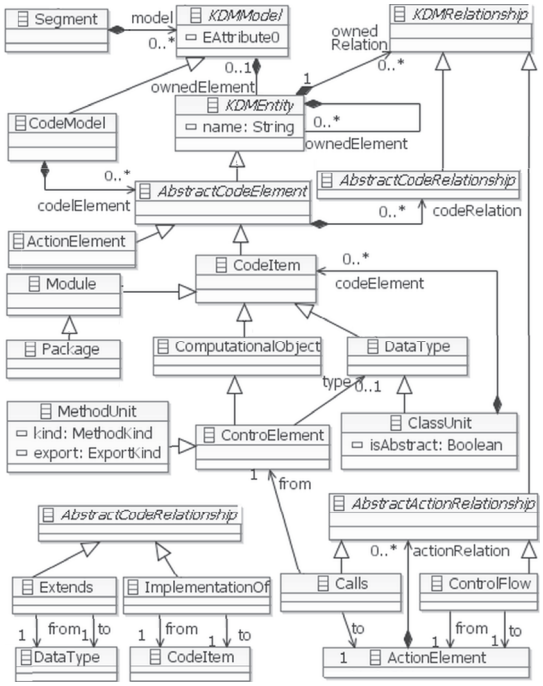

Figure 3: KDM metamodel.

To obtain the KDM model from the code model generated in the first stage, we use the KDM discoverer provided by MoDisco, although it was necessary to adapt it. This discoverer, implemented as a M2M ATL transformation, creates models that conform to KDM from the Java model. This transformation was enhanced to obtain the required information to recover sequence diagrams. The transformation provided by MoDisco is not fully specified, there are elements in the Java model that are not fully transformed into their corresponding KDM elements which results in loss of information. Some considerations are the followings:

- in a method invocation expression, arguments of the method that are simple variables, such as local variables or parameters, are missing in the KDM model;
- in a method invocation expression, if the object on which the method is invoked is a simple variable, it is not present in the corresponding KDM model and therefore the relationship between this variable and the method invocation is also missing;
- in infix expressions, operands that are simple variables are missing in the KDM model.

To solve these problems we added new ATL helpers and modified some rules of the discoverer provided by MoDisco. The outcome of this transformation applied to the *eLib* program model is the KDM model partially depicted in Figure 4.

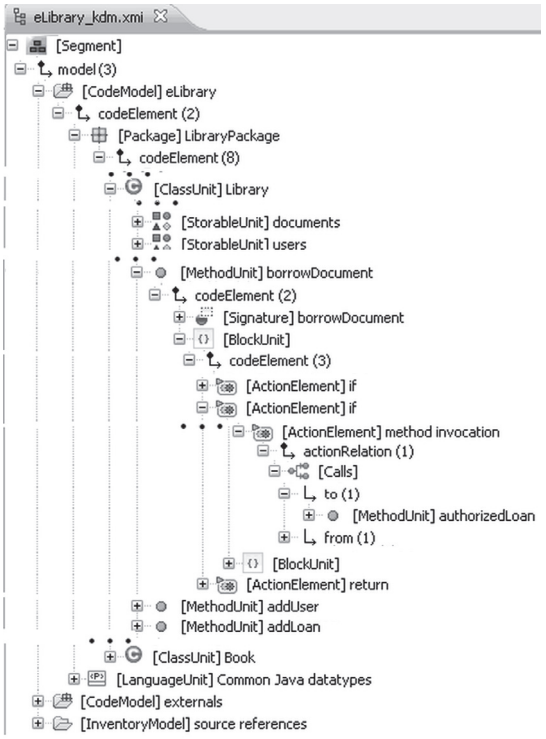

Figure 4: KDM model of the *eLib* program.

The KDM model consists of one *Segment* that owns three models each representing one architectural view of the system. The model *eLibrary* owns one instance of *Package* called *LibraryPackage*. It contains eight instances of *UnitClass* that represent user-defined classes in the program *eLib* such as *Library* and *Book*. The *ClassUnit* Library owns *StorableUnits* that represent variables and *MethodUnits* that represent member functions. The *MethodUnit borrowDocument* owns one *Signature* that represents the procedure signature and one *BlockUnit* that represents logically and physically related blocks of *ActionElements* (basic unit of behavior), for instance blocks of statements. The *BlockUnit* contain *ActionElements*

such as *if*, *method invocation* and *return* statements.

4.2.2 Recovering PIM Model

The KDM model of the *eLib* program is the starting point to recover PIM models by means of a KDM to UML discoverer implemented as an ATL M2M transformation that takes an input model conforming to KDM and produces an output model conforming to UML. The source metamodel corresponds to KDM, partially shown in Figure 3. The target metamodel corresponds to the UML metamodel of interactions that is partially shown in Figure 5.

An Interaction owns lifelines, messages and fragments. A lifeline represents an individual participant in the interaction. A message defines a communication between lifelines. A fragment may be an *Interaction, an ExecutionSpecification* or *OccurrenceSpecification*. An *ExecutionSpecification* specifies the execution of a unit of behavior or action within the lifeline. *OccurrenceSpecifications*, ordered along a lifeline, are the basic semantic unit of interactions. The sequences of occurrences specified by them are the meanings of Interactions. A *GeneralOrdering* represents a binary relation between two occurrence specifications which describes that one occurrence specification must occur before the other in a valid trace.

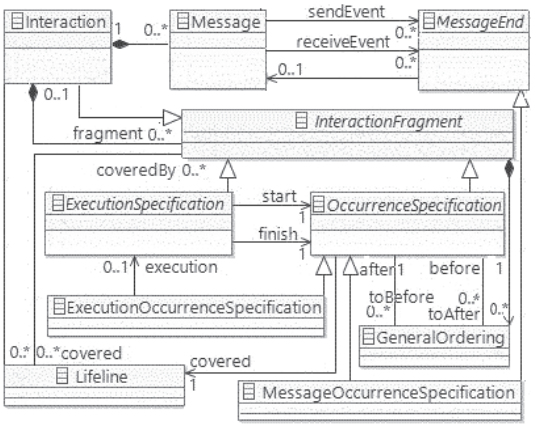

Figure 5: UML metamodel of interaction diagram.

```
module kdm2interaction;
create OUT : UML from IN : KDM;
-- ------------------------------------ R U L E S ------------------------------------
rule method2Interaction {
from m:MM!MethodUnit (m.is_relevantPublicMethod())
to interact:MM1!Interaction (
   name     <- m.get_ContainerClass().name + '::' + m.name
  ,lifeline <- thisModule.createLifeline(m)
  ,lifeline <- m.get_Variables()->collect(v|thisModule.object2Lifeline(v))
  ,message  <- m.get_calls()
  ,generalOrdering <- thisModule.CreateGeneralOrdering_Initial(m)
  ,generalOrdering <- m.get_pairsOfCalls()->collect ( p | thisModule.CreateGeneralOrdering(p->at(1)))
  ,generalOrdering <- thisModule.CreateGeneralOrdering_Finish(m)            )
}
rule methodInvocation2message {
from call:MM!Calls (call.is_relevantCall()) to msj:MM1!Message (
  name <- if(call.to.kind= #constructor)then call.get_ContainerClass().name+ '::'+'create_'+ call.to.name
                                       else call.get_ContainerClass().name + '::' + call.to.name endif
  ,argument <-call.get_Arguments()->collect(name|thisModule.createArgument(name))
  ,sendEvent<- send
  ,receiveEvent <- receive )
  ,send: MM1!MessageOccurrenceSpecification (...)
  ,receive: MM1!MessageOccurrenceSpecification(...)
}
unique lazy rule createLifeline {
from m:MM!MethodUnit     to obj:MM1!Lifeline (
  name <- m.get_ContainerClass().name.toLower()+': ' +m.get_ContainerClass().name
 ,coveredBy <-thisModule.createActionExecutionSpecification(m,m.get_ContainerClass().name.toLower()))
}
unique lazy rule object2Lifeline {
from d:MM!DataElement
to obj:MM1!Lifeline ( name <- d.name  + ': ' + d.type.name  )
}
unique lazy rule CreateGeneralOrdering {
from call:M1!Calls  to genOrdering: MM1!GeneralOrdering(
   name <- call.to.name + '->' + call.get_ContainerMethod().nextTo(call).to.name
  ,before <- thisModule.resolveTemp(call, 'receive')
  ,after<- thisModule.resolveTemp(call.get_ContainerMethod().nextTo(call),'send') )
}
unique lazy rule createActionExecutionSpecification {
from m:MM!MethodUnit ,name:MM!StringType        to exeSpec:MM1!ActionExecutionSpecification ()...}
```

Figure 6: *KDM2interaction* transformation.

The transformation *KDM2interaction,* partially depicted in Figure 6, specifies the way to produce interaction diagrams (target model) from KDM models (source model). Source and target models must conform to the KDM metamodel and the UML metamodel respectively. The most relevant rules that carry out the transformation are described below.

The rule *method2Interaction* transforms each public method that is relevant into an interaction whose name is formed by the method name preceded by the class name that owns that method. The first lifeline corresponds to the object that is an instance of the class that owns the method. The other lifelines are obtained from the local variables and parameters of the method that are object references on which a method is invoked. The messages of the interaction are obtained from the calls of the method. A partial order between the messages is stated by *generalOrderings* created from lazy rules. The fragments owned by the interaction are set in the other rules by means of the opposite links.

The rule *methodInvocation2message* transforms each instance of *Calls* into a message. If the method invocation kind is 'constructor', the message name is the method name preceded by the string 'create'. In all other cases, the message name is the invoked method name preceded by the name of the class that owns this method. This rule states the sender and the receiver of the message.

The rule *CreateGeneralOrdering* creates an instance of *GeneralOrdering* from a call. Its name is formed by the message name corresponding to the call followed by the method name corresponding to the next call, separated by '->' indicating the order between the messages. *Before* and *after*, instances of *MessageOccurrenceSpecification*, specify the order between them.

The rule *createLifeline* creates a lifeline from the class that owns the method from which the interaction is created. The rule *object2Lifeline* creates a lifeline from a variable.

The rule *createActionExecutionSpecification* creates a control focus (execution occurrence) from a method and a lifeline name. The *start* and the *finish*, instances of *ExecutionOccurrence-Specification* are created from lazy rules.

Figure 7 shows the model resulting from the transformation *KDM2interaction* when it is applied to the KDM model corresponding to *eLib* program, in particular the model of the sequence diagram that represents the message interchange among objects triggered by the execution of the method *borrowDocument* inside the class Library.

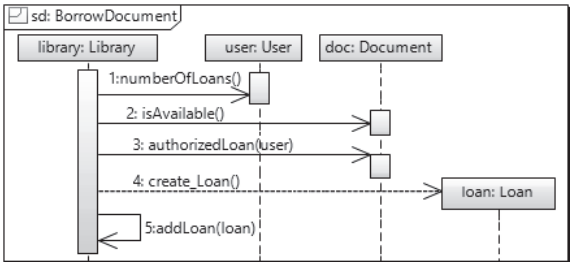

Figure 7: The sequence diagram.

5 ANALYSIS OF RESULTS

The proposed recovery process allows obtaining interaction diagrams that describe invocation resolution for relevant methods, that is to say, complex methods that involve many method invocations. It makes no sense to draw just one huge diagram for the whole functioning of the system because the size may exceed the cognitive abilities of human even for small systems. Therefore, it is better to split it up according to the most important sub-computations and focus the view on the most important methods, thus following the natural approach to the construction of these diagrams (Tonella and Potrich, 2005). Besides, to simplify the diagram, only the calls made directly from the method of interest are resolved as shown in Figure 7.

We validated our approach by using the open source application platform Eclipse EMF that is the core technology in Eclipse for model-driven development. The Eclipse-MDT MoDisco project provide model discoverers, generators and transformation languages such as ATL, we used them to discover sequence diagrams from Java code.

Our results can also be considered as a contribution to MoDisco community since they do not provide support for reverse engineering sequence diagrams. To achieve this, the MoDisco KDM Discoverer was extended in order to obtain the required information for recovering sequence diagrams and other interaction diagrams.

The example used along this paper, the Java program *eLib*, allowed us to compare the results with the ones obtained in (Tonella and Potrich, 2005), thereby validating our approach.

6 CONCLUSION AND FUTURE WORK

This paper proposes an approach for software

modernization based on the integration of traditional reverse engineering techniques to the ADM context. We describe how to extract higher-level models expressed in terms of UML diagrams from object-oriented code. We believe that ADRE approach provides benefits with respect to processes based only on traditional reverse engineering techniques. Interoperability between the tools and services of the adherents of the standards is facilitated achieving well-defined interfaces and well-defined formats for interchange of information about software models used by the software modernization tools.

Our case study shows how to recover sequence diagrams that describe invocation resolution for relevant methods obtaining diagrams that only show the calls made directly from the method of interest. Currently, we are extending the *KDM2interaction* transformation to obtain more complete diagrams.

At present, we are testing our approach on a Customer Relationship Management (CRM) application. We are adapting an existing desktop CRM application to mobile platforms achieving interoperability with multiple mobile platforms. The idea is to use different layers of abstraction that can map a 'write once' application into native executable programs that will run on multiple platforms. To achieve this, a metamodel for the Haxe language was specified.

Other future activities in reverse engineering should push towards a tight integration of dynamic analysis and human feedback into automatic reverse engineering techniques. The idea is to learn from expert feedback to automatically produce results.

REFERENCES

ADM 2014. Architecture-Driven Modernization Task Force. http://www.omgwiki.org/admtf/doku.php

ASTM 2011. Abstract Syntax Tree Metamodel, version 1.0, OMG Document Number: formal/2011-01-05. http://www.omg.org/spec/ASTM

ATL 2014. Atlas Transformation Language (ATL). http://www.eclipse.org/atl/documentation/

Barbier, F., Deltombe, G., Parisy, O., and Youbi, K. 2011. Model Driven Engineering: Increasing Legacy Technology Independence. In *2nd India Workshop on Reverse Engineering in The 4th India Software Engineering Conference* (pp. 5-10). India: CSI ed.

Blu Age Reverse Modeling 2014. http://bluage.com /en/en_product/en_ba_rev_modeling.html

Brambilla, M., Cabot, J., and Wimmer, M. 2012. *Model-Driven Software Engineering in Practice.* Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers.

Cánovas Izquierdo, J., and García Molina, J. 2009a. A domain specific language for extracting models in software modernization. *Model Driven Architecture - Foundations and Applications,* Lecture Notes in Computer Science, 2009 (Volume 5562, pp. 82-97). Berlin, Heidelberg: Springer-Verlag.

Cánovas Izquierdo, J., and García Molina, J. 2009b. Extracción de modelos en una modernización basada en ADM. *Actas de los Talleres de las Jornadas de Ingeniería de Software y BBDD*, (Vol 3, issue 2, pp. 41-50). http://www.sistedes.es/ficheros/actas-talleres-JISBD/Vol-3/No-2/DSDM09.pdf

Deissenboeck, F., and Ratiu, D. 2006. A Unified Meta Model for Concept-Based Reverse Engineering. In 3rd *International Workshop on Metamodels, Schemes, Grammars, and Ontologies for Reverse Engineering*. http://planet-de.org/atem2006/atem06Proceedings.pdf

Java Call Tracer 2014. http://sourceforge.net/projects/java calltracer/

KDM 2011. Knowledge Discovery Metamodel, version 1.3, OMG Document Number: formal/2011-08-04. http://www.omg.org/spec/KDM/1.3

MDA 2014. The Model-Driven Architecture. http://www.omg.org/mda/

Martinez, L., Favre, L., and Pereira C. 2013. Architecture-Driven Modernization for Software Reverse Engineering Technologies. In *Progressions and Innovations in Model-Driven Software Engineering*. IGI Global, pp 288-307.

MoDisco 2014. Model Discovery. http://www.eclipse.org/ MoDisco

MOF 2011. Meta Object Facility (MOF) Core Specification Version 2.4.1, formal/2011-08-07. http://www.omg.org/spec/MOF/2.4.1

QVT 2011. QVT: MOF 2.0 Query, View, Transformation. Version 1.1, OMG Document Number: formal/2011-01-01. http://www.omg.org/spec/QVT/1.1/

Tonella, P., and Potrich, A. 2005. Reverse Engineering of Object Oriented Code. Monographs in Computer Science. Heidelberg: Springer-Verlag.

UML 2011. Unified Modeling Language: Infrastructure. Version 2.4.1, OMG Specification formal/2011-08-05. http://www.omg.org/spec/UML/2.4.1/

Visual Paradigm 2014. http://www.visual-paradigm.com/solution/visualtrace/