Recovering Use Case Diagrams from Object Oriented Code: An MDA-based Approach

Conference Paper · April 2011

DOI: 10.1109/ITNG.2011.130 · Source: DBLP

3 authors:

Claudia Pereira
Universidad Nacional del Centro de la Provincia de Buenos Aires
33 PUBLICATIONS **109** CITATIONS
SEE PROFILE

Liliana Martinez
Universidad Nacional del Centro de la Provincia de Buenos Aires
32 PUBLICATIONS **108** CITATIONS
SEE PROFILE

Liliana Favre
Universidad Nacional del Centro de la Provincia de Buenos Aires
82 PUBLICATIONS **383** CITATIONS
SEE PROFILE

Recovering Use Case Diagrams from Object Oriented Code: an MDA-based Approach

Claudia T. Pereira (1), Liliana I. Martinez (1), and Liliana M. Favre (1,2)

(1) Universidad Nacional del Centro de la Provincia de Buenos Aires (Argentina)
(2) Comisión de Investigaciones Científicas de la Provincia de Buenos Aires (Argentina)
{cpereira, lmartine, lfavre}@exa.unicen.edu.ar

ABSTRACT

Modernization of legacy systems requires the existence of technical frameworks for information integration and tool interoperability that allow managing new platform technologies, design techniques and processes. MDA (Model Driven Architecture), adopted by the OMG (Object Management Group), is aligned with this requirement. Reverse engineering techniques play a crucial role in system modernization. In light of these issues, this article describes a framework to reverse engineering MDA models from object oriented code. This framework distinguishes three different abstraction levels linked to models, metamodels and formal specifications. At model level, transformations are based on static and dynamic analysis. At metamodel level, transformations are specified as OCL (Object Constraint Language) contracts between MOF (Meta Object Facility) metamodels which control the consistency of these transformations. The level of formal specification includes algebraic specifications of MOF metamodels and metamodel-based transformations. This article shows how to reverse engineering use case diagrams from Java code in the MDA context focusing on transformations at model and metamodel levels. We validate our approach by using Eclipse Modeling Framework, Ecore metamodels and ATL (Atlas Transformation Language).

Keywords: legacy systems, metamodeling, model driven development, reverse engineering, transformation

1- INTRODUCTION

Reverse engineering is the process for analyzing available software artifacts such as requirements, design or code with the objective of extracting information and providing high-level views on the underlying system [1].

At the beginning, reverse engineering was focused mainly on recovering high-level architecture or diagrams from procedural code to deal with problems such as comprehending data structures or databases. At that time, static analysis techniques based on compiler theory and abstract interpretation were developed. Later, when object oriented languages emerged, a growing demand of reverse engineering systems appeared and the focus of software analysis moved from static to dynamic analysis. With the emergency of UML (Unified Modeling Language) [2, 3], reverse engineering was focused on how to extract higher level views of the system expressed by different kinds of diagrams.

Nowadays, software and system engineering industry evolves to manage new platform technologies, design techniques and processes. New technical frameworks for information integration and tool interoperability such as MDD (Model Driven Development) have created the need to develop new analysis tools and specific techniques. MDD refers to a range of development approaches based on the use of software models as first class entities, being one of them MDA (Model Driven Architecture) [4].

The key idea behind MDA is to separate the specification of the system functionality from its implementation on specific platforms, managing the software evolution from abstract models to implementations increasing the degree of automation and achieving interoperability with multiple platforms, formal languages and programming languages. MDA distinguishes different kinds of models: CIM (Computation Independent Model), PIM (Platform Independent Model), PSM (Platform Specific Model) and ISM (Implementation Specific Model). In MDA, artifacts generated during software development are represented using common metamodeling languages. The essence of MDA is MOF (Meta Object Facility) [5] metamodel that allows different kinds of artifacts from multiple vendors to be used together in the same project. MOF provides two metamodels EMOF (Essential MOF) and CMOF (Complete MOF). EMOF favors simplicity of implementation over expressiveness. CMOF is a metamodel used to specify more sophisticated metamodel. EMF (Eclipse Modeling Framework) was created for facilitating system modeling and the automatic generation of Java code [6]. It started as an implementation of MOF resulting Ecore, which is a meta-metamodel for describing models in EMF. EMF has evolved starting from the experience of the Eclipse community to implement a variety of tools and to date is highly related to Model Driven Engineering.

With the emergence of MDA, new approaches should be developed to reverse engineering PIMs and PSMs from object oriented code. This article contributes a framework to reverse engineering MDA models from object oriented code based on the integration of metamodeling, static analysis and dynamic analysis techniques and formal specification. In a previous work, we show how to reverse engineering PSMs including class diagrams, behavioral diagrams and state diagrams [7]. In this article, the emphasis is given to reverse engineering of PIMs, we describe how to recover use case diagrams from Java code in particular. We show how MOF metamodels can be used to analyze the consistency of model recovery processes. Also, we show how to integrate our results into EMF by using the ATL (Atlas Transformation

Language), a model transformation language in the field of Model Driven Engineering that is developed on top of the Eclipse platform [8].

This article is organized as follows. Section 2 deals with related work and CASE tools related to reverse engineering. Section 3 describes a framework for MDA-based reverse engineering. Section 4 describes code-to-model transformation based on static and dynamic analysis and shows how to recover use case diagrams from Java code. Section 5 specifies reverse engineering process as metamodel-based transformations, and shows an ATL transformation between Ecore metamodels. Finally, Section 6 highlights conclusions.

2- RELATED WORK

An overview of the state-of-the-art of reverse engineering techniques may be found in [9]. Canfora and Di Penta [10] carried out a survey of existing work in the area of reverse engineering that compares existing work, discusses success and provides a road map for future developments in the area.

Some works have contributed to reverse engineering use case diagrams from object oriented code. How to discover basic use cases by specifying the beginning methods of method calling sequences of the dynamic information is shown in [11]. Authors utilize rules to mine the relations among basic use cases to build the use case Diagrams. Di Luca, Fasolino and De Carlini [12] identify use cases by analyzing class method activation sequences triggered by input events and terminated by output events. This approach produces diagrams at different levels of abstraction.

A relevant overview of techniques that have been recently investigated and applied in the field of reverse engineering of object oriented code is provided in [13]. Authors describe the algorithms involved in the recovery of UML diagrams from code and some of the techniques that can be adopted for their visualization. The algorithms deal with the reverse engineering of class, object, interaction, state and package diagrams. The underlying principle in this approach is that information is derived statically by performing propagation of proper data in a data flow graph.

Software industry evolves to manage new platform technologies, design techniques and processes. There is an increased demand of modernization 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 interoperation like MDA.

Many works are linked to MDD-based reverse engineering. An approach to bridging legacy systems to MDA that includes an architecture description language and a reverse engineering process is presented in [14]. A method that implements model driven transformations between particular platform-independent (business views) and platform-specific (IT architectural) models is described in [15]. At PIM level, they use business process models and at PSM level, the IT architectural models are service-oriented and focus on specific platform using Web service and workflows. Gueheneuc [16] proposes a study of class diagram constituents with respect to their recovery from object oriented code. In [17] MOMENT is described, a rigorous framework for automatic legacy system migration in MDA. A project that assesses the feasibility of applying MDD to the evolution of a legacy system is reported in [18]. The first steps towards the definition of a metamodel that unifies a conceptual view on programs with the classical structure-based reverse engineering metamodels is shown in [19]. A feasibility study in reengineering legacy systems based on grammars and metamodels is described in [20].

OMG is involved in the definition of standards to successfully modernize existing information systems. The OMG ADM (Architecture-Driven Modernization) Task Force is developing a set of specifications and promoting industry consensus on modernization of existing applications [21].

2-1 CASE TOOLS AND REVERSE ENGINEERING

When UML emerged, a new problem was how to extract higher level views of the system expressed by different kind of diagrams. The reverse engineering tool RevEng extracts UML diagrams from C++ code such as class, object, state, sequence, collaboration and package diagrams [13].

A tool-assisted way of introducing models in the migration towards MDA is described in [22]. Authors propose to automatically extract architecturally significant models (called Container models) and subsequently refactoring them to achieve models in higher-level of abstraction.

The Fujaba Tool Suite project is suited to provide an easy to extend UML, Story Driven Modeling and Graph Transformation platform with the ability to add plug-ins [23]. The tool combines UML static diagrams and UML behavior diagrams (Story Diagrams). Furthermore, it supports the generation of Java code. Fujaba is configured with plug-ins for Reverse Engineering and Design Pattern recognition.

EMF [6] was created for facilitating system modeling and the automatic generation of Java code. EMF started as an implementation of MOF resulting Ecore, the EMF metamodel comparable to EMOF. EMF has evolved starting from the experience of the Eclipse community to implement a variety of tools and to date is highly related to MDE (Model Driven Engineering). For instance, ATL is a model transformation language in the MDE field that is developed on top of the Eclipse platform. The ATL Integrated Development Environment provides a number of standard development tools (syntax highlighting, debugger, etc.) that aim to ease the design of ATL transformations.

Commercial tools such as IBM Rational Software Architect, Spark System Enterprise Architect or Together are integrated with Eclipse-EMF [24].

Few MDA-based CASE tools support QVT (Query/View/Transformation) or at least, any of the QVT languages [25]. As an example, IBM Rational Software Architect and Spark System Enterprise Architect do not implement QVT. Other tools partially support QVT, for instance Together allows defining and modifying transformations M2M (model-to-model) and M2T (model-to-text) that are QVT-Operational compliant. Blu Age [26] provides a generic and extensible approach to model extraction and discovery from multiple types of legacy systems. It uses a model-based approach and a metamodel-driven methodology.

Medini QVT partially implements QVT [27]. It is integrated with Eclipse and allows the execution of transformations expressed in the QVT-Relation language. Eclipse M2M, the official tool compatible with of Eclipse 3.5 and EMF 2.5.0, is still under development and implements the specification of QVT-Operational. MoDisco [28] provides an extensible framework to develop model-driven tools to support use-cases of existing software modernization. It uses EMF to describe and manipulate models, M2M to implement transformation of models into other models, Eclipse M2T to implement generation of text and Eclipse Java Development Tools.

2-2 OUR CONTRIBUTION

Tonella and Potrich [13] describe a reverse engineering approach of object oriented code. Authors adapted concepts and algorithms of data flow analysis described in [29] to obtain UML diagrams from Java code, particularly class diagrams, object diagrams, interaction diagrams, state diagrams and package diagrams. The approach proposed in [13] is based on the techniques developed during a collaboration work at CERN (Conseil Européen pour la Recherche Nucléaire) for the experiments to be run on the Large Hadron Collider. Through these experiments, the approach was validated to about half million lines of object oriented code.

Although authors of the approach mentioned above do not reverse engineering use case diagrams, this process can take place applying the same principles. In this article, we complement this work to reverse engineering use case diagrams from Java code. Following the same ideas and starting only from object-oriented code we propose a process to automatically retrieve use case diagrams, including use cases and their relationships (dependency and inheritance). The proposed process also assists in identifying the type of dependency, *extend* or *include*, task that can not be performed fully automatically. The dynamic analysis of the code would indicate the possibility of a relationship *extend*, but the final decision must be made by an expert.

Moreover, we propose a formalization of the recovery processes described at [13] in terms of standards involved in MDA such as MOF metamodels and, we analyze how to recover PIMs including use cases.

We exemplify reverse engineering of use case diagrams from Java code and contribute a process that allows extending the functionality of the existing MDA Case tools. This article also describes experiments in EMF by using ECORE metamodels and ATL.

3- MDA-BASED REVERSE ENGINEERING

To reverse engineering MDA models from object oriented code, we propose a framework based on the integration of compiler techniques, metamodeling and formal specification. This framework distinguishes different abstraction levels linked to models, metamodels and formal specifications (Fig. 1).

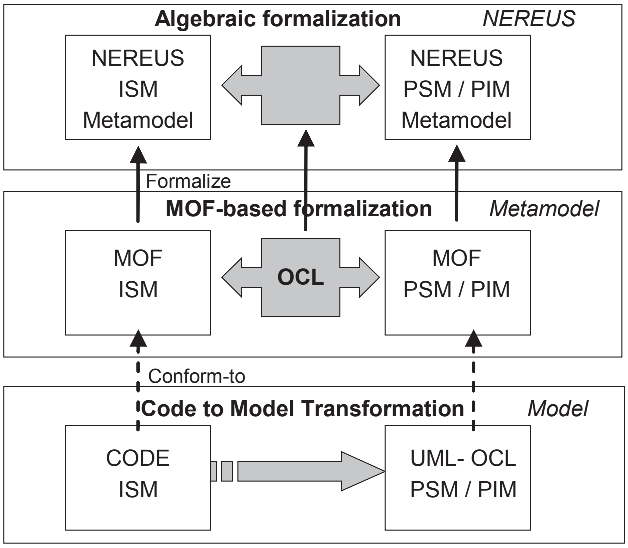


Figure 1 Framework for MDA-based reverse engineering

The model level includes code, PIMs and PSMs. PIMs and PSMs are expressed in UML and OCL [30]. The subset of UML diagrams useful for PSMs includes class diagrams, object diagrams, state diagrams, interaction diagrams and package diagrams. A PIM can be expressed by means of use case diagrams, activity diagrams, interaction diagrams to model system processes and state diagrams to model lifecycle of the system entities.

At model level, transformations are based on classical compiler construction techniques. They involve processes with different degrees of automation, which can go from totally automatic static analysis to processes that require human intervention, to dynamically analyze the resultant models. All the algorithms that deal with reverse engineering share an analysis framework. The basic idea is to describe source code or models by an abstract language and perform a propagation analysis in a data-flow graph called in this context object-data flow. This static analysis is complemented with dynamic analysis supported by tracer tools.

The metamodel level includes MOF metamodels that describe the transformations at model level. MOF metamodel uses an object modeling framework that is essentially a subset of UML core [2]. The modeling concepts are classes, associations, data types and packages. At this level, MOF metamodels describe families of ISMs, PSMs and PIMs. Transformations are specified as OCL contracts between a source metamodel and a target metamodel. MOF metamodels “control” the consistency of these transformations.

The level of formal specification includes specifications of MOF metamodels and metamodel-based transformations in the metamodeling language NEREUS [31] that can be used to connect them with different formal and programming languages. NEREUS is suited for specifying metamodels such as MOF that are based on the concepts of entity, association and system. Favre [31] defines a bridge between MOF metamodels and NEREUS and shows how to integrate NEREUS with CASL (Common Algebraic Specification Language) [32, 33].

This article focuses on reverse engineering of PIMs from object oriented code at model and metamodel level. A process for the recovery of use case diagrams from Java code at PIM level is analyzed and specified as metamodel-based transformations.

4- CODE-TO-MODEL TRANSFORMATION

At model level, transformations rely on static and dynamic analysis (Fig.2). Static analysis extracts static information that describes the structure of the software reflected in the software documentation, whereas dynamic analysis information describes the structure of the run-behavior. Static information can be extracted by using techniques and tools based on compiler techniques such as parsing and data flow algorithms [29]. Dynamic information can be extracted by using debuggers, event recorders and general tracer tools.

An MDA-based reverse engineering process abstracts MDA models from ISMs. This process involves different phases. First, the source code is parsed to obtain an abstract syntax tree associated with the source programming language grammar. Next, a metamodel extractor extracts a simplified, abstract version of a language that ignores all instructions that do not affect the data flows, for instance all control flows such as conditionals and loops. The information represented according to this metamodel allows building the data-flow graph for a given source code, as well as conducting all other analysis that do not depend on the graph. The idea is to derive statically information by performing a propagation of data. Different kinds of analysis propagate different kinds of information in the data-flow graph extracting the different kinds of diagrams that are included in a model.

The static analysis relies on classical compiler techniques [29] and abstract interpretation [34]. Data-flow graph and the generic flow propagation algorithms are specializations of classical flow analysis techniques [29]. Abstract interpretation allows obtaining automatically as much information as possible about program executions without having to run the program on all input data and then ensuring computability or tractability. These ideas were applied for optimizing compilers.

In reverse engineering process, static analysis can be enriched with dynamic analysis that relies on an execution model generated by execution tracer tools. The execution model includes the following components: a set of objects, a set of attributes for each object, a location and value of an object type for each object, and a set of messages. Additionally, types such as Integer, String, Real and Boolean are available for describing types of attributes and parameters of methods or constructors.

The static and dynamic information could be shown as separated views or merged in a single view. The dynamic behavior could be visualized as an execution scenery which describes interaction between objects. To extract specific information, it is necessary to define particular views of these sceneries. Although the construction of these views can be automated, their analysis requires some manual processing in most cases.

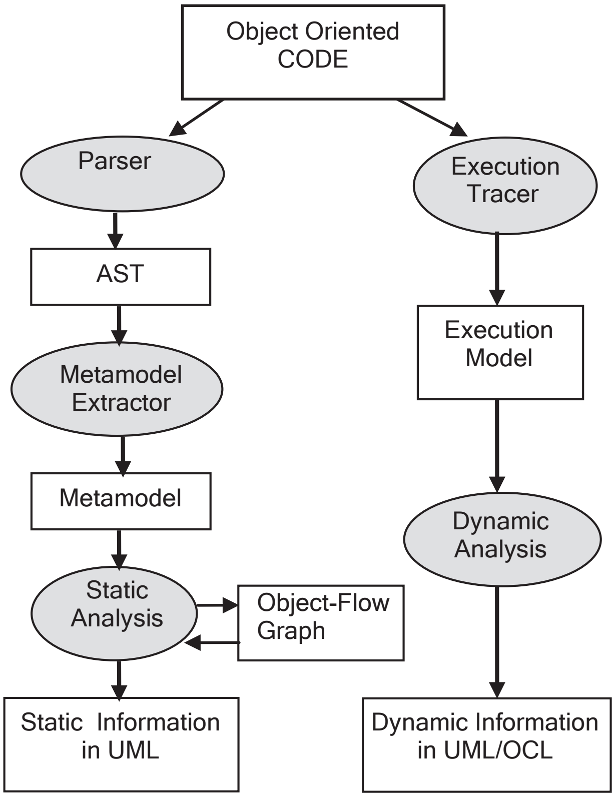


Figure 2 Reverse engineering at model level

4-1 RECOVERING USE CASE DIAGRAMS

A reverse engineering approach of object oriented code is described in [13]. Authors adapted concepts and algorithms of data flow analysis described in

[29] to obtain UML diagrams from Java code. In this article, we complement this work to reverse engineering use case diagrams from Java code in the MDA context. Following the approach proposed in [13], in the first stage of the diagram recovery process, the Java language is simplified into an abstract language where all features related to the object flow are maintained while the other syntactic details are dropped (Fig. 3). The choice of this program representation is motivated by the computational complexity and the "nature" of the object oriented programs whose code is typically structured so as to impose more constraints on the data flows than on the control flows. For example, the sequence of method invocations may change when moving from an application which uses a class to another one, while the possible ways to copy and propagate object references remains more stable.

The obtained abstract code is the base for the recovery process of use cases. A program P consists of zero or more occurrences of declarations (D), followed by zero or more statements (S). Declarations are of three types: attribute declarations, method declarations and constructor declarations. Statements are of three types: allocation statement, assignment statement and method invocations.

```
P ::=  D * S *

D ::=  a                               a: class attribute name
     | m ( $f_1$, ..., $f_k$ )         m: method name;
     | cs ($f_1$, ..., $f_k$ )         cs: class constructor
                                       f1,…,fk: formal parameters

S ::=  x = new c ( $a_1$, ...,$a_k$);  c: class name;
     | x = y;                          $a_1$, ...,$a_k$: actual parameters
     | [ x =]  y.m( $a_1$, ...,$a_k$); x, y: program location
```

Figure 3 Abstract syntax of the simplified Java language

To recover use cases which model the intended behavior of the system, the following diagram elements are required [3]:

1. Use case: it is a description of a set of sequences of actions that a system performs to provide an observable result of value to an actor. Basic use cases are extracted analyzing public methods. Each public method of a class corresponds to a use case whose name is the same as the method name.
2. Generalization: it is a relationship where the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent. The child may be substituted in any place the parent appears. For each pair of same-name methods which are

members of different classes with a generalization between them, a generalization between use cases corresponding to these methods is generated. To determine the right kind of the relationship, either *redefine* or *enrichment*, the expertise of the software engineer is needed.

3. Dependency: it is a using relationship between two use cases in which a change to one (the independent use case) of them may affect the semantics of the other (the dependent use case). For each public method of a class, the method calls are analyzed to extract dependence relationships among basic use cases. Each message sent to an object corresponds to a dependence between use cases. The kind of relationship, *include* or *extend*, may be inferred by a dynamic analysis.
4. Actor: represents a coherent set of roles that users of use cases play when interacting with these use cases. Actors can be human or they can be automated systems. Although the actors are used in the models, they are not actually part of the system. They live outside the system. Hence, actors can not be automatically inferred.

The main steps to reverse engineering use case diagrams are the following:

1. generate abstract code from java code,
2. execute the algorithm for static recovery,
3. determinate the kind of dependence relationship between use cases from dynamic analysis,
4. define high-level use case diagrams.

Next, we describe these steps and exemplify in terms of the same study case used in [13], which exemplifies reverse engineering by using the Java program *eLib* that supports the main library functions. It supposes an archive of documents of different categories, properly classified. The documents are of different kinds, including books, journals and technical reports. Each of them has specific functionality. Each document can be uniquely identified and library users can request document for loan. In order to borrow a document, a user must be identified by the librarian. As regards the loan management, users can borrow documents up to a maximum number. 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. A detailed description may be found at [13].

Step 1. Generate abstract code from Java code maintaining all features related to the object flow. Fig. 4.a partially shows Java code of the *eLib* program, the method *borrowDocument* of class *Library*, *numberOfLoans* of class *User*, the methods *isAvailable* and *authorizedLoan* of class *Document* and *authorizedLoan* of classes *Journal* and *TechnicalReport*. Fig. 4.b shows the translation to the abstract code.

```
class Library {  …
public boolean borrowDocument
                        (User user, Document doc)
  { if (user ==null) || doc ==null) return false;
     if (user.numberOfLoans()< MAX_LOANS
            && doc.isAvailable() &&
             doc.authorizedLoan(user))

               Loan loan=new Loan(user, doc)

                  addLoan(loan);
                  return true;  }
       return false;}    …}

class User {
  public int numberOfLoans()  ...}

class Document {
    public boolean isAvailable() …
    public boolean authorizedLoan
                               (User user) …}

class Journal extends Document {
  public boolean authorizedLoan
                               (User user)…}

class TechnicalReport extends Document {
  public boolean authorizedLoan(User user)
    …}
```

a. Java code of *eLib* program

```
Library.borrowDocument
         (Library.borrowDocument.user,
            Library.borrowDocument.doc)
Library.borrowDocument.user.numberOfLoans();
Library.borrowDocument.doc.isAvailable();
Library.borrowDocument.doc.authorizedLoan
          (Library.borrowDocument.user);
Library.borrowDocument.loan=new Loan
          (Library.borrowDocument.user,
           Library.borrowDocument.doc)
Library.borrowDocument,this.addLoan
          (Library.borrowDocument.loan);


User.numberOfLoans()


Document.isAvailable()
Document.authorizedLoan
        (Document.authorizedLoan.user)


Journal.authorizedLoan
        (Journal.authorizedLoan .user)


TechnicalReport.authorizedLoan
        (TechnicalReport.authorizedLoan.user)
```

b. Abstract code of *eLib* program

Figure 4 From Java to Abstract Code

Step 2. Execute the algorithm for static recovery (Fig. 5). It is executed on the abstract code and returns three sets containing use cases, dependences and generalizations. These sets are statically determined and allow obtaining relevant information to recover diagrams.

Once these sets have been initialized as empty sets, the first step of the algorithm is to obtain the use case set. For each public method in the abstract code, a new use case is generated and included in the set. If there are methods with the same name, each method name is prefixed by the class name.

The second step is to obtain the dependence set. For each expression '*p.g()*' included in a public method *m* on the abstract code, if *g* is a public method, a dependence relationship between the use cases corresponding to *m* and *g* is included in the set. For instance, the *borrowDocument* abstract code contains the expression *'doc.isAvailable()'*, *isAvailable* is a public method, then a dependence relationship between the use cases corresponding to *borrowDocument* and *isAvailable* is included in the set.

```
-- initialization of sets
  useCases ←{ };
  dependences ← { };
  generalizations ←{ }

-- generating useCases set
  for each public method m
      useCases ← useCases ∪ { m }
  endfor

-- generating dependences set
  for each expr: 'p.g()' / g is a public method
      m  ← method (scope(expr))
      dependences ← dependences ∪ { (m, g) }
  endfor

-- generating generalizations set
  for each  $m_i$ in useCases
      for each $m_k \neq m_i$ in useCases
          if name ($m_k$) = name($m_i$)
              if class ($m_k$)  is parent of class ($m_i$)
                      generalizations ← generalizations ∪ { ($m_i$,$m_k$) }
                  else
                  if class ($m_i$)  is parent of class ($m_k$)
                      generalizations ←generalizations ∪ { ($m_k$,$m_i$)}
                      endif
                  endif
          endif
      endfor
  endfor
```

Figure 5 Recovering Use Case diagrams – Algorithm

Next, the algorithm obtains the generalization set. For each pair of same-name use cases that correspond to methods of different classes that have a generalization between them, a generalization between the use cases corresponding to these methods is generated. For instance, there is a generalization between the use cases derived from *authorizedLoan* methods of the *Journal* and *Document* classes.

Fig. 6 shows use cases, dependences and generalizations obtained by the algorithm to recover use case diagrams applied to abstract code of *eLib* program shown in Fig 4.b.

useCases	Dependences	Generalizations
borrowDocument	numberOfLoans isAvailable Document_ authorizedLoan	
numberOfLoans		
isAvailable		
Document_authorizedLoan		
Journal_authorizedLoan		Document_ authorizedLoan
TechnicalReport_authorizedLoan		Document_ authorizedLoan

Figure 6 Algorithm output

Step 3. Determinate the kind of dependence relationship between use cases in the dependence set by means of the information obtained from dynamic analysis. Each initial use case obtained during static analysis corresponds to a method. For each of these methods, the system is executed to recover all possible threads, primary flow and alternative paths:

a. The common sub-thread reflects primary flow and allows detecting *include* relationships between use cases

b. The others sub-threads (alternative paths) may correspond to *extend* relationships. To decide between *extend* and *include* alternatives will be necessary to analyze the source code of the method from which begins the sub-threads. The software engineer is required to accomplish this task.

For example, the different execution threads corresponding to the method borrowDocument are the following:

Thread 1: borrowDocument –> numberOfLoans –> isAvailable –> authorizedLoan

Thread 2: borrowDocument –> numberOfLoans –> isAvailable

Thread 3: borrowDocument –> numberOfLoans

Thread 4: borrowDocument

These threads represent alternative paths that might indicate *extend* relationships. To decide between *extend* and *include* alternatives is necessary to analyze the source code of the method borrowDocument. In this particular case, the analysis of the code indicates that the kind of relationships is *include*.

Step 4. Define high-level use case diagrams. The proposed recovery process allows obtaining use case diagrams that correspond to a low-level functional view. To obtain higher-level views, use cases may be clustered in single abstract use cases by applying simple heuristics [12]:

a. A group of use cases, where one of them includes exclusively the remaining ones, will be clustered in a single more abstract use case shown in a separate diagram.

b. A group of use cases, where a use case is extended by the remaining ones, will be clustered in a single more abstract use case shown in a separate diagram.

The obtained information from the reverse engineering process allows building use case diagrams. Fig. 7 shows the use case diagram that corresponds to the Java code of Fig. 4.a.

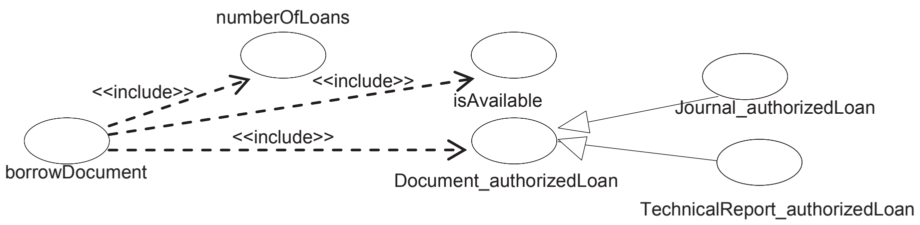


Figure 7 Use Case Diagram

The high-level use case diagrams are obtained by applying the heuristics described in the step 4 of the reverse engineering process. Considering the use case diagram shown in fig 7, if the use case borrowDocument included exclusively the use cases numberOfLoans, isAvailable and Documet_authorizedLoan, the group of use cases would be clustered in a single more abstract use case called borrowDocument.

The proposed process focuses on reverse engineering use case diagrams, including use cases and their relationships (dependency and inheritance). The process also assists in identifying the type of dependency, *extend* or *include.* Our approach might be completed with the additional steps that depend heavily on human intervention such as detect duplicated use cases, to eliminate trivial use case, identify actors, define use-case package in order to structure the use cases model by dividing it into smaller parts.

5- SPECIFYING REVERSE ENGINEERING IN MDA

A metamodeling technique is used to specify reverse engineering in MDA. MOF metamodels are used to describe the transformations at model level. For each transformation, source and target metamodels are specified. A source metamodel defines the family of source models to which transformation can be applied. A target metamodel characterizes the generated models.

This approach was implemented in the Eclipse Modeling Framework (Fig. 8). Source and target metamodels conform to Ecore metamodel, which is comparable to EMOF. The transformations between models are specified in ATL. It is an hybrid language that provides a mix of declarative and imperative constructs. ATL mainly focuses on the model to model transformations which can be specified by means of ATL modules. An ATL module is composed of the following elements:

- a header section that defines the names of the transformation module and the variables of the source and target metamodels.
- an optional import section that enables to import some existing ATL libraries
- a set of helpers that can be used to define variables and functions.
- a set of rules that defines how source model elements are matched and navigated to create and initialize the elements of the target models.

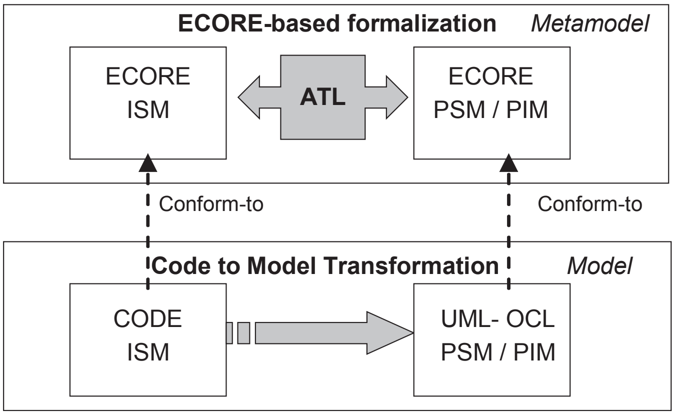
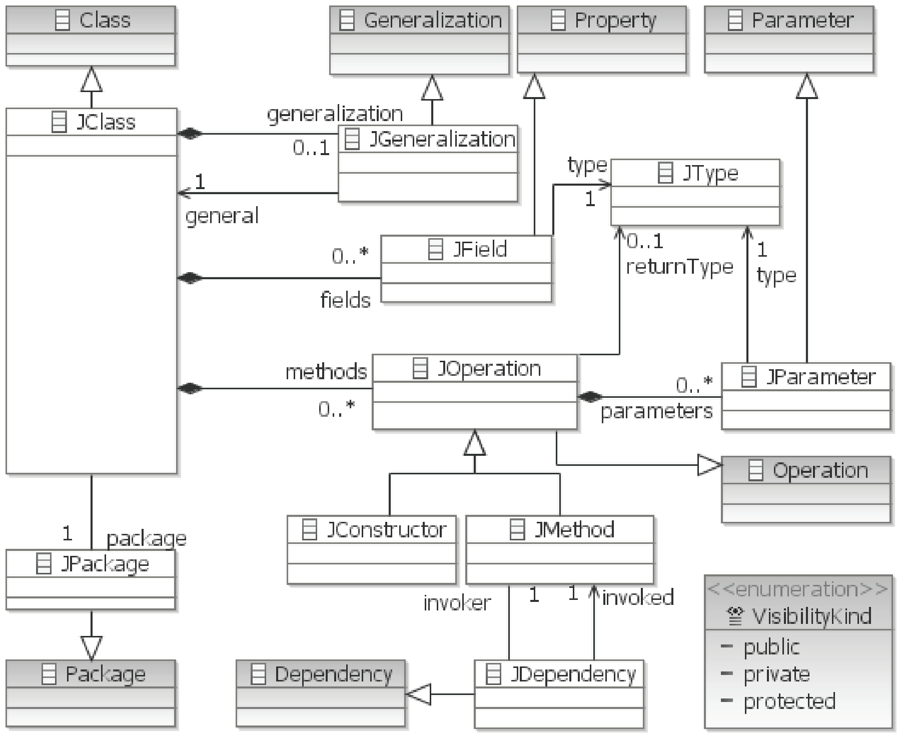


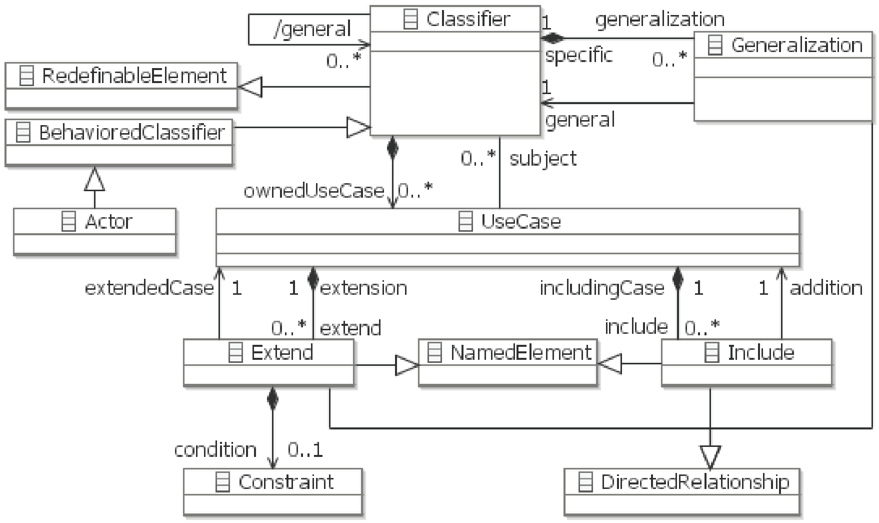
Figure 8 Transformations in Eclipse

5-1 SPECIFYING REVERSE ENGINEERING OF USE CASES

To specify reverse engineering of use case diagrams at metamodel level, source and target metamodels are specified (Fig. 9). Both metamodels conforms to metametamodel Ecore. The source metamodel corresponds to the Java language which is a specialized UML metamodel. Fig. 9.a partially shows this metamodel, the shaded metaclasses correspond to metaclasses of the UML metamodel, whereas the remainder correspond to the specialization. It includes constructs that represent Java classes, which own fields and operations. The target metamodel corresponds to the use cases UML metamodel (Fig. 9.b). A UseCase is a kind of behaviored classifier that specifies some behavior that the subject can perform in collaboration with one or more actors. A use case may be related to others through generalization, extend or include relationships.



a. Specialized UML metamodel of Java language



b. UML metamodel of Use Case Diagram

Figure 9 Java to UseCases Transformation - Source and target metamodels

Model transformation is specified in ATL (Fig. 10). The module *Java2UseCases* that corresponds to the transformation specifies the way to produce use cases diagrams (target model) from Java code (source model). Both source and target models must conform to their respective metamodel. MM corresponds to Ecore metamodel of JavaCode and MM1 corresponds to Ecore metamodel of UML.

```
module Java2UseCases;
create OUT : MM1 from IN : MM;

helper context  MM!JClass
      def : allParents () : Sequence(MM!JClass) =
            ...   ;

helper context  MM!JMethod
      def : equalSignature (met: MM!JMethod)  : Boolean =
      self.name = met.name  and
      self.parameters->size() = met.parameters->size()and
      self.parameters->forAll(p1 |
            met.parameters->exists(p2| p1.type = p2.type)) ;

helper context  MM!JMethod
      def : redefinableMethod ()  : MM!JMethod =
      self.class.allParents().methods ->select (m|
            m.oclIsTypeOf(MM!JMethod)   and
            self.equalSignature(m))->first() ;

rule Method2UseCase{
from m:MM!JMethod
      (m.visibility = #public and
      not m.class.generalization.oclIsUndefined() )
to uc:MM1!UseCase (
      name <- m.class.name + '_' + m.name,
      clientDependency <- m.dependences,
      general <-   m.redefinableMethod())
}

rule Method2UseCase1{
from m:MM!JMethod
      (m.visibility = #public
      and  m.class.generalization.oclIsUndefined() )
to uc:MM1!UseCase (
      name <- m.class.name + '_' + m.name,
      clientDependency <- m.dependences )
}

rule Dependency2Dependency{
from d:MM!JDependency
      (d.invoked.visibility = #public
      and d.invoker.visibility = #public)

to use_case_dep:MM1!Dependency (
      client <- d.invoker ,
      supplier <- d.invoked   )
}
```

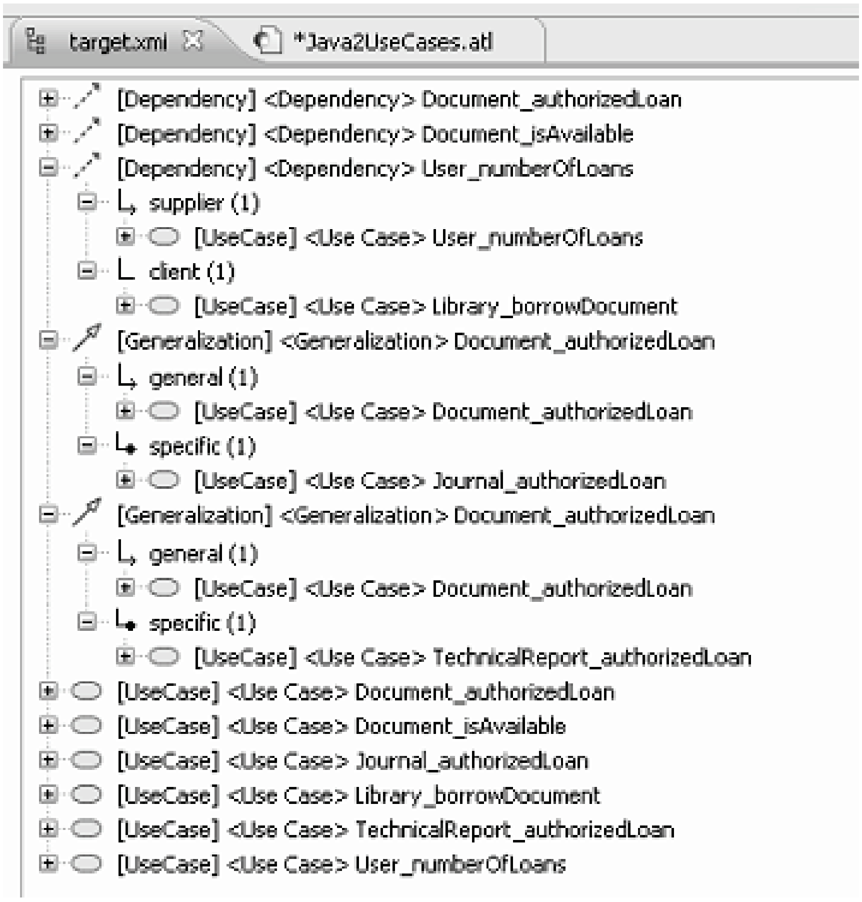
Figure 10 Java to UseCases ATL transformation

The module is composed of helpers and rules. The main rules are the followings. The rule Method2UseCase transforms each public method, which can redefine an inherited method, into a use case that can have dependences and generalizations.

The rule Method2UseCase1 transforms each public method, which does not redefine inherited method, into a use case that can only have dependences.

The rule Dependency2Dependency transforms each dependency between two public methods into a dependency between the use cases corresponding to these methods.

Fig.11 shows the target model resulting of the application of the transformation from the Java code of the *eLib* program (Fig. 4.a). It is written in XMI (XML Metadata Interchange), an OMG standard that combines XML, MOF, and UML for integrating tools, repositories, and applications in distributed heterogeneous environments.



```
[Dependency] <Dependency> Document_authorizedLoan
[Dependency] <Dependency> Document_isAvailable
[Dependency] <Dependency> User_numberOfLoans
    supplier (1)
        [UseCase] <Use Case> User_numberOfLoans
    client (1)
        [UseCase] <Use Case> Library_borrowDocument
[Generalization] <Generalization> Document_authorizedLoan
    general (1)
        [UseCase] <Use Case> Document_authorizedLoan
    specific (1)
        [UseCase] <Use Case> Journal_authorizedLoan
[Generalization] <Generalization> Document_authorizedLoan
    general (1)
        [UseCase] <Use Case> Document_authorizedLoan
    specific (1)
        [UseCase] <Use Case> TechnicalReport_authorizedLoan
[UseCase] <Use Case> Document_authorizedLoan
[UseCase] <Use Case> Document_isAvailable
[UseCase] <Use Case> Journal_authorizedLoan
[UseCase] <Use Case> Library_borrowDocument
[UseCase] <Use Case> TechnicalReport_authorizedLoan
[UseCase] <Use Case> User_numberOfLoans
```

Figure 11 Target model resulting from the transformation

6- CONCLUSION

This article describes a framework to reverse engineering MDA models based on the integration of static and dynamic analysis, metamodeling and formal specification. This work focuses on transformations at model and metamodel levels in particular.

The main contribution of this article is a recovery process of use case diagrams from object-oriented code by applying static and dynamic analysis and its formalization in terms of transformations based on MOF metamodels. Although we analyze the reverse engineering of Java code, the bases of our approach can be easily applied to other object-oriented language.

The process to recover use cases is based on techniques for reverse engineering of UML diagrams described in [13] which were validated in a large-scale project. Therefore, our work could be considered as an extension of [13] that allows recovering use case diagrams.

Additionally we show an MDA-based formalization. We propose a metamodeling technique based on Ecore metamodels and the ATL transformation language. We implement and validate our approach in the Eclipse Modeling Framework.

Little support for reverse engineering use cases from code is provided for the existing CASE tools. We foresee integrate our results in the MDA-based Case tools.

REFERENCES

[1] Sommerville, Software Engineering, Addison Wesley, 2004.

[2] UML Infrastructure. OMG Specification formal/2010-05-03. Retrieved January 2012 from http://www.omg.org/spec/UML/2.3/Infrastructure/PDF.

[3] UML Superstructure. OMG Specification formal/2010-05-05. Retrieved January 2012 from http://www.omg.org/spec/UML/2.3/Superstructure/PDF

[4] MDA. The Model Driven Architecture. http://www.omg.org/mda (last accessed January 2012).

[5] MOF. MOF 2.0. OMG Specification formal/2006-01-01. Retrieved January 2012 from http://www.omg.org/spec/MOF/2.0/PDF.

[6] Eclipse. The eclipse modeling framework. http://www.eclipse.org/emf (last accessed January 2012).

[7] L. Favre, L. Martinez, and C. Pereira, “MDA-based Reverse Engineering of Object-Oriented Code”, 14 th Int. Conf. EMMSAD 2009, LNBIP, 29. Heidelberg: Springer-Verlag, pp. 251-263, 2009.

[8] ATL Documentation. http://www.eclipse.org/atl/documentation (last accessed January 2012).

[9] L. Angyal, L. Lengyel, and H. Charaf, “An Overview of the State-of-the-Art Reverse Engineering Techniques”, Proc. 7th International Symposium of Hungarian Researchers on Computational Intelligence, pp. 507–516, 2006.

[10] G. Canfora and M. Di Penta, “New Frontiers of Reverse Engineering. Future of Software engineering”, Proc. of Future of Software Engineering (FOSE 2007). Los Alamitos:IEEE Press. pp. 326-341, 2007.

[11] Q. Li, S. Hu, P. Chen, L. Wu, and W. Chen, “Discovering and Mining Use Case Model in Reverse Engineering”, Int. Conf. on Fuzzy Systems and Knowledge Discovery (FSKD 2007), pp. 431 – 436, 2007.

[12] G. Di Luca, A. Fasolino, and U. De Carlini, “Recovering Use Case models from Object-Oriented Code: a Thread-based Approach”, Seventh Working Conf. on Reverse Engineering (WCRE 2000), pp.108-117, 2000.

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

[14] B. Qiao, H. Yang, W. Chu, and B. Xu, “Bridging Legacy Systems to Model Driven Architecture”, Proc. 27th Annual Int. Computer Aided Software and Applications Conf. Los Alamitos: IEEE Press, pp. 304-309, 2003.

[15] J. Koehler, R. Hauser, S. Kapoor, F. Wu and S. Kumaran, “A Model-Driven Transformation Method”. Proc. Seven IEEE Enterprise Distributed Object Computing Conference, (EDOC, 2003), pp. 186–197, 2003.

[16] Y. Gueheneuc. “A Systematic Study of UML Class Diagram Constituents for their Abstract and Precise Recovery”. Proc.11th Asia-Pacific Software Engineering Conference (APSEC 2004), Los Alamitos: IEEE Computer Society, pp. 265-274, 2004.

[17] A. Boronat, J. Carsi and I. Ramos, “Automatic Reengineering in MDA using Rewriting Logic a Transformation Engine”, Proc. Ninth European Conference on Software Maintenance and Reengineering (CSMR´05), Los Alamitos: IEEE Computer Society, pp. 228-231, 2005.

[18] A. MacDonald, D. Russell, and B. Atchison, “Model Driven Development within a Legacy System: An Industry Experience Report”, Proc. the 2005 Australian Software Engineering Conference (ASWEC 05), Los Alamitos: IEEE Press, pp. 14-22, 2005.

[19] F. Deissenboeck and D. Ratiu, “A Unified Meta Model for Concept-Based Reverse Engineering”, Proc. 3rd International Workshop on Metamodels, Schemes, Grammars, and Ontologies for Reverse Engineering. 2006. Retrieved January 2012 from http://planet-mde.org/atem2006/atem06Proceedings.pdf

[20] T. Reus, H. Geers and A. van Deursen, “Harvesting Software System for MDA-based Reengineering”, Lecture Notes in Computer Science, Heidelberg: Springer-Verlag, vol. 4066, pp. 220-236, 2006.

[21] ADM. ADM Task Force, Standards Roadmap. http://adm.omg.org/ (last accessed January 2012).

[22] N. Mansurov and D. Campara, “Managed Architecture of Existing Code as a Practical Transition Towards MDA”, Lecture Notes in Computer Science Heidelberg: Springer-Verlag, vol. 3297, pp. 219-233, 2005.

[23] FUJUBA tool. http://www.fujaba.de/ (last accessed January 2012).

[24] MDA CASE tools. http://case-tools.org/mda.html (last accessed January 2012).

[25] QVT: MOF 2.0 Query, View, Transformation, Formal/2008-04-03. Retrieved January 2012 from http://www.omg.org/spec/QVT/1.0/

[26] Blu Age Tool. http://case-tools.org/tools/blu_age.html (last accessed January 2012)

[27] Medini QVT. http://projects.ikv.de/qvt (last accessed January 2012).

[28] Modisco. http://www.eclipse.org/MoDisco (last accessed January 2012).

[29] A. Aho, R. Sethi, and J. Ullman, Compilers: Principles, Techniques, and Tools (2nd ed.), Addison-Wesley, 1985.

[30] OCL, Version 2.3.1, OMG: formal/2012-01-01. Retrieved January 2012 from http://www.omg.org/spec/OCL/2.3.1/PDF.

[31] L. Favre, “A Formal Foundation for Metamodeling”, Reliable Software Technologies, ADA Europe 2009, LNCS 5570, Springer-Verlag, pp.177-191, 2009.

[32] L. Favre, Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution. Engineering Science Reference, 1 Edition. USA, 2010.

[33] M. Bidoit and P. Mosses, CASL User Manual - Introduction to Using the Common Algebraic Specification Language, LNCS 2900, Springer-Verlag, 2002.

[34] N. Jones and F. Nielson, “Abstract Interpretation: A Semantic Based Tool for Program Analysis”, Handbook of Logic in Computer Science, Vol. 4. Clarendon Press, Oxford, pp. 527–636, 1995.