IV International Conference on Computational Methods for Coupled Problems in Science and Engineering
COUPLED PROBLEMS 2011
M. Papadrakakis, E. Oñate and B. Schrefler (Eds)

MODERNIZING SCIENCE&ENGINEERING SOFTWARE SYSTEMS

LILIANA FAVRE[*], LILIANA MARTINEZ[†] AND CLAUDIA PEREIRA[†]

[*] Universidad Nacional del Centro de la Provincia de Buenos Aires
Comisión de Investigaciones Científicas de la Provincia de Buenos Aires
Tandil, Argentina
e-mail: lfavre@exa.unicen.edu.ar

[†] Universidad Nacional del Centro de la Provincia de Buenos Aires
Tandil, Argentina
email: {cpereira, lmartine@exa.unicen.edu.ar}

Key words: Reverse Engineering, Model Driven Architecture, Software Modernization, Modeling, Reliability.

Abstract. As the demands for modernized legacy systems rise, so does the need for frameworks for information integration and tool interoperability. The Object Management Group (OMG) has adopted the Model Driven Architecture (MDA), which is an evolving conceptual architecture that aligns with this demand. MDA could help solve coupling problems of multidisciplinary character in science and engineering that consist of one or more applications, supported by one or more platforms. The objective of this paper is to describe rigorous techniques to control the evolution from science & engineering software legacy systems to MDA technologies. We propose a rigorous framework to reverse engineering code in the context of MDA. Considering that validation, verification and consistency are crucial activities in the modernization of systems that are critical to safety, security and economic profits, our approach emphasizes the integration of MDA with formal methods.

1 INTRODUCTION

Nowadays, software industry is evolving and tackling new approaches aligned with Internet, object orientation, distributed components, new modeling languages and new platforms. However, the majority of the large engineering applications running today in many research organizations were developed many years ago with technology that is now obsolete. These old systems, known as legacy systems, are still business-critical. New platforms and applications must interoperate with legacy software systems. Their complete replacement is dangerous and their maintenance is increasingly expensive. That is the reason for the demand of modernization of legacy system to extend their useful lifetime [18].

As the demands for modernized legacy systems rise, so does the need for frameworks for information integration and tool interoperability. The Object Management Group (OMG) has adopted the Model Driven Architecture (MDA), which is an evolving conceptual architecture that addresses the challenges of networked and changing system environments. MDA could help solve coupling problems of multidisciplinary character in science and engineering that consist of one or more applications, supported by one or more platforms [14].

The outstanding ideas behind MDA are separating the specification of the system functionality from its implementation on specific platforms, managing the software evolution from abstract models to implementations. MDA shifted the center of software development from code to models. In the MDA context, code is considered as an artifact derived from model transformations. The concepts of model, metamodel and model transformation are central in MDA. Models play a major role in MDA which distinguishes at least the following models:

• Platform Independent Model (PIM): a model with a high level of abstraction that is independent of any implementation technology.

• Platform Specific Model (PSM): a tailored model to specify the system in terms of the implementation constructs available in one specific platform.

• Implementation Specific Model (ISM): a description (specification) of the system in source code.

The initial diffusion of MDA was focused on its relation with UML as modeling language [20,21]. However, there are UML users who do not use MDA, and MDA users who use other modeling languages such as Domain Specific Languages (DSL). The essence of MDA is MOF (Meta Object Facility) metamodel that allows different kinds of software artifacts to be used together in a single project [15]. The basic idea is to create a common specification for communication between applications. The MOF 2.0 Query, View, Transformation (QVT) metamodel is the standard for expressing transformations [17].

Validation, verification and consistency are crucial activities in the modernization of legacy systems that are critical to safety, security and economic profits. Reasoning about models of systems is well supported by automated theorem provers and model checkers, however these tools are not integrated into CASE tools environments.

The objective of this paper is to describe rigorous techniques to control the evolution from science &engineering software legacy systems to MDA technologies. We propose a rigorous framework to reverse engineering code in the context of MDA that emphasizes the integration of techniques related to MDA, such as metamodeling, with formal methods.

The paper is organized as follows. In the next section we provide background material on modernization of systems and Case tools. The next four sections consist of the main ideas of our work. Section 3 describes a framework for reverse engineering. Section 4 describes how to transform code into models through static and dynamic analysis. Section 5 and Section 6 describe how to formalize reverse engineering processes in terms of MOF metamodels and formal specifications respectively. Finally, conclusions are drawn in Section 7.

2 RELATED WORK

The article [4] compares existing work, discusses success and provides a road map for possible future developments in the area. The reengineering of a deteriorated object oriented industrial program written in C++ is described in [9]. In order to deal with this problem, the authors designed and implemented several restructuring tools and used them in specific reengineering scenarios. A variety of techniques for object oriented reengineering based on patterns are distinguished in [7].

Many works are linked to MDD-based reverse engineering. A rigorous framework for automatic legacy system migration in MDA called MOMENT is described in [3]. A tool-assisted way of introducing models in the migration towards MDA is presented in [12]. The article [13] describes model-based dynamic analysis techniques that relate system execution traces and its models such as testing whether a system run satisfies a property that a certain model specifies and measuring how various model features materialize in a system run.

The material presented in [19] is based on techniques developed during a collaboration with CERN (Conseil Européen pour la Recherche Nucléari) in the introduction of tools for software quality assurance, among which a reverse engineering tool called RevEng was presented. This tool extracts UML diagrams (class diagrams, object diagrams, state diagrams, sequence and collaboration diagrams and package diagrams) from C++ code.

Modernizing large industrial software systems is impossible without appropriate tool support. All of the MDA tools are partially compliant to MDA features. They provide good support for modeling and limited support for automated transformation in reverse engineering.

Techniques that currently exist in MDA CASE tools provide little support for validating models in the design stages. Reasoning about models of systems is well supported by automated theorem provers and model checkers, however these tools are not integrated into CASE tools environments. Another problem is that as soon as the requirements specifications are handed down, the system architecture begins to deviate from specifications. Only research tools provide support for formal specification and deductive verification. Many CASE tools support reverse engineering, however, they only use more basic notational features with a direct code representation and produce very large diagrams.

To be able to reason about software systems, MDA CASE tools need a common information base aligned to MOF. However few MDA-based CASE tools support MOF and QVT or at least, any of the QVT languages. 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 model-to-model (M2M) and model-to-text (M2T) that are QVT-Operational compliant [5].

The Eclipse Modeling Framework (EMF) [8] 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 Model Driven engineering (MDE). Commercial tools such as IBM Rational Software Architect, Spark System Enterprise Architect or Together are integrated with Eclipse-EMF [5].

3 A RIGOROUS FRAMEWORK FOR REVERSE ENGINEERING

We propose to reverse engineering MDA models from object oriented code starting from the integration of compiler techniques, metamodeling and formal specification. Figure 1 shows a framework for reverse engineering that integrates static and dynamic analysis, metamodeling and formal specification. It distinguishes three different abstraction levels linked to models, metamodels and formal specifications.

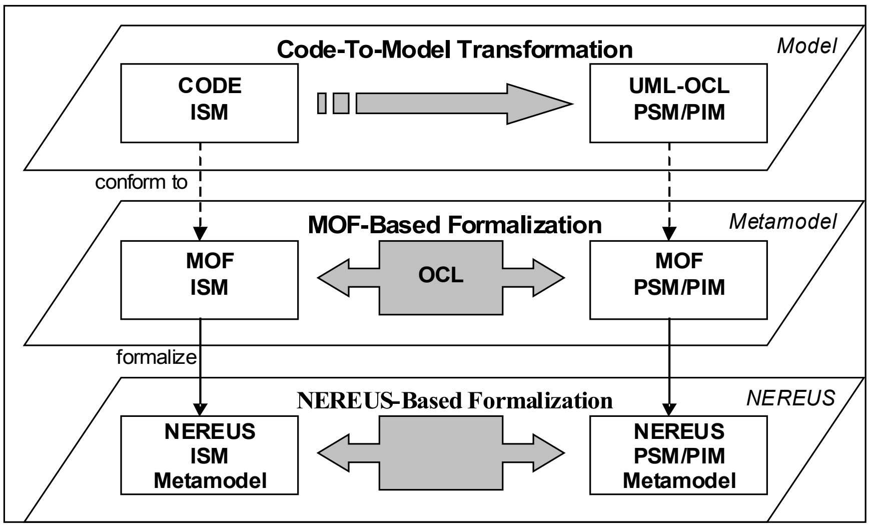


Figure 1. MDA-based reverse engineering

The model level includes code, PIM and PSM. A PIM is a model with a high level of abstraction that is independent of an implementation technology [14]. A PSM is a tailored model to specify a system in terms of specific platform such J2EE or .NET. PIM and PSM are expressed in UML and OCL [20,21,16]. The subset of UML diagrams that are useful for PSM includes class diagrams, object diagrams, state diagrams, interaction diagrams and package diagrams. On the other hand, a PIM can be expressed by means of use case diagrams, activity diagrams, interactions diagrams to model system processes and, state diagrams to model lifecycle of the system entities. An ISM is a specification of the system in source code.

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 human intervention requiring processes to dynamically analyze the resultant models. All the algorithms that deal with the 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 [15]. A metamodel is an explicit model of the constructs and rules needed to construct specific models. MOF metamodels use an object modeling framework that is essentially a subset of UML 2.3 core [20]. The modeling concepts are classes which model MOF metaobjects, associations, which model binary relations between metaobjects, data types which model other data, and packages which modularize the models. At this level MOF metamodels describe families of ISM, PSM and PIM. Every ISM, PSM and PIM conforms to a MOF metamodel. Metamodel transformations are specified as OCL contracts between a source metamodel and a target metamodel. MOF metamodels “control” the consistency of these transformations.

The formal specification level includes specifications of MOF metamodels and metamodel transformations in the metamodeling language NEREUS that can be used to connect them with different formal and programming languages [10,11].

To sum up, at model level, instances of ISM, PSM and PIM are generated by applying static and dynamic analysis. Static analysis builds an abstract model of the state and determines how the program executes to this state. Dynamic analysis operates by executing a program and evaluating the execution traces of the program. Contracts based on MOF-metamodels “control” the consistency of these transformations and NEREUS facilitates the connection of the metamodels and transformations with different formal languages.

Our work could be considered as an MDA-based formalization of the process described by Tonella and Potrich in [19]. Additionally, we propose a different algorithm for extracting UML State diagrams and new processes to recover PIM including use case diagrams and activity diagrams. We also propose to include OCL specifications (preconditions, postconditions and invariants) in UML diagrams. Other contributions are linked to the automation of the formalization process and interoperability of formal languages [10,11].

The following sections describe reverse engineering at three different abstraction levels corresponding to code-to-model transformation, MOF-metamodel formalization and algebraic formalization.

4 RECOVERING MODELS FROM CODE

At model level, transformations are based on static and dynamic analysis. Static analysis extracts static information that describes the software structure reflected in the software documentation (e.g., the text of the source code) whereas dynamic analysis information describes the structure of the run-behavioral. Static information can be extracted by using techniques and tools based on compiler techniques such as parsing and data flow algorithms. Dynamic information can be extracted by using debuggers, event recorders and general tracer tools.

We suppose that the reverse engineering process starts from an ISM that could reflect, for instance, the migration of legacy code to object oriented code. The first step in the migration towards MDA is the introduction of PSMs. Then, a PIM is abstracted from the PSMs omitting platform specific details.

Next, we describe the process for recovery PSMs from code. Figure 2 shows the different phases. The source code is parsed to obtain an abstract syntax tree (AST) associated with the source programming language grammar. Then, 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 conditional 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 PSM.

The static analysis is based on classical compiler techniques and abstract interpretation [1]. On the one hand, data-flow graph and the generic flow propagation algorithms are specializations of classical flow analysis techniques [1]. On the other hand, 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 to optimizing compilers.

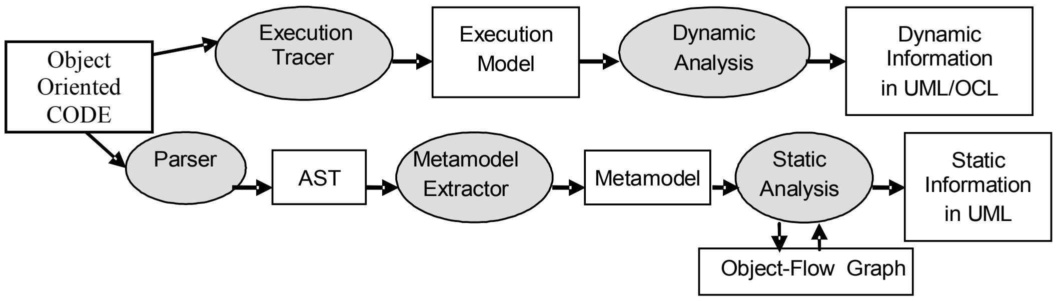


Figure 2. From code to models: Static and dynamic analysis

The static and dynamic information could be shown as separated views or merged in a single view. In general, the dynamic behavior could be visualized as execution sceneries which describe 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.

In the MDA context, we can distinguish code-based dynamic analysis and model-based dynamic analysis. The first is based on an execution model including 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. On the other hand, model-based dynamic analysis is based on model-level debugging and evolution.

4.1 Recovering Class Diagrams

A class diagram is a representation of the static view that shows a collection of static model elements, such as classes, interfaces, methods, attributes, types as well as their properties (e.g., type and visibility). Besides, the class diagram shows the interrelationships holding among the classes [21].

Reverse engineering of class diagrams from code is a difficult task that cannot be automated due to certain elements in the class diagram carry behavioral information that cannot be inferred just from the analysis of the code. By analyzing the syntax of the source code, internal class features such as attributes and methods and their properties (e.g. the parameters of the methods and visibility) can be recovered. From the source code, associations, generalization, realizations and dependencies may be inferred too. However, to distinguish between aggregation and composition we need to capture system states through dynamic analysis. Figure 3 shows relationships that can be detected statically between a C++ program and a UML class diagram.

Dynamic analysis allows generating execution snapshot to collect life cycle traces of object instances and reason from tests and proofs. Execution tracer tools generate execution model snapshots that allow us to deduce complementary information, for instance information to detect compositions. A composition is a particular aggregation in which the lifetime of the part is controlled by the whole (directly or transitively) and we can identify it by generating tests and scanning dependency configurations between the birth and the death of a part object according to those of the whole. In the same way, the execution traces of different instances of the same class or method could guide the construction of invariants or preconditions and postconditions respectively.

C++ Code	*Class attribute*
	
	class A {
	…B b; }
		Parameter
		class A {
		void f(B b) {…b.g(…);}}
		
		Local variable
		class A {
		void f (..) { B b; …b.g(..);}}
			Inheritance
			class A: *access-specifier* B
			{…}
			
			access-specifier = public, private or protected

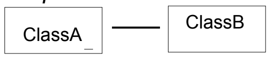
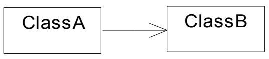
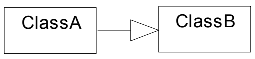
UML relationship	*Association/Aggregation/ Composition* 	*Dependency* 	*Generalization* 

Figure 3. From C++ code to UML class diagram

4.2 Recovering State Diagrams

A state transition diagram describes the life cycle of objects that are instances of a class from the time they are created until they are destroyed. Object state is determined by the value of its attributes and possibly by the variables involved in attribute computations. The basic elements of a state diagram are states, identified as equivalence classes of attribute values, and transitions triggered by method invocation.

Our approach to recover state diagrams has similar goals to abstract interpretation that allows obtaining automatically as much information as possible about program executions without having to run it on all input data and then ensuring computability or tractability. These ideas were applied to optimizing compilers, often under the name data-flow analysis [1]. In our context, an abstract interpretation performs method invocation using abstract domains instead of concrete attribute values to deduce information about the object computation on its actual state from the resulting abstract descriptions of its attributes. This implies to abstract equivalence classes that groups attribute values corresponding to the different states in which the class can be and the transitions among state equivalence classes.

Then, the first step is to define an appropriate abstract interpretation for attributes (which give the state of the object) and modifier class method (which give the transitions from state to state to be represented in the state diagram).

The recovery algorithm iterates over the following activities: the construction of a finite automata by executing abstract interpretations of class methods and the minimization of the automata for recovering approximate state equivalence classes. To ensure tractability, our algorithm proposes an incremental minimization every time a state is candidate to be added to the automaton. When it is detected that two states are equivalents, they are merged in an only state. This could lead to modification of the parts of the automaton that had been previously minimized. To optimize the comparison of pairs of states, these are classified according to their emerging transitions. Let m be a bound of the number of transformer methods of a class, the idea is to generate subsets of the set of transformer methods. The subset of emerging transitions of a new state belongs, in a particular snapshot, to one of them. Two states are candidates to be equivalent if they belong to the same subset. Then, it is sufficient to compare all the pairs composed by the state and one element of the subset. Considerable human interaction to select which abstract interpretations should be executed is required [6].

As an example, Figure 4 (a) shows a diagram including states (s_1, s_2,..,s_8) and transitions (m_1,m_2,…,m_6). Fig. 4 (b) shows a simplified snapshot of the automaton when a transition to s_5 is added. Then, the shaded states could belong to the same equivalence state class. s_8 belongs to the same subset of s_4 and an equivalence analysis is carried out concluding that s_8 and s_4 can be merged. Figure 4 (c) (d) (e) shows the successive transformations.

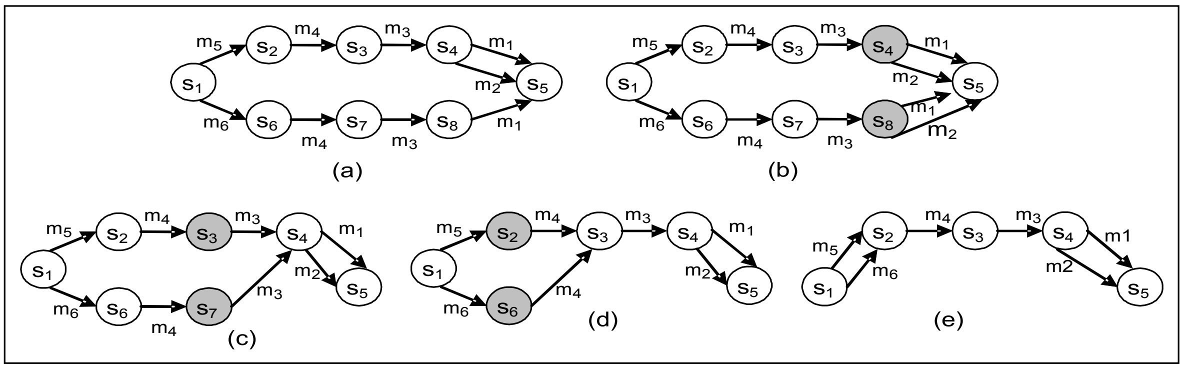


Figure 4. Recovering minimum State Diagram

A C++ implementation was developed to test the feasibility of our recovering algorithm. In the following, the algorithm for the identification of the states is described:

```
/* initialization of  different sets*/
set-of-states initialStates = {};     /* states defined by class constructors*/
set-of-states pendingStates ={};  /* set of states pending of analysis*/
set-of-states allStates = {};         /* set of all states*/
/*definition of initial states for the objects of the class*/
for each class constructor c
        /*executing an abstract interpretation of each class constructor*/
        state s = abstractInterpretationState (c, {});
        initialStates = initialStates  ∪ {s};
        pendingStatesPending = pendingStates ∪ {s};
        allStates = allStates ∪{s}
endfor
set-of-transitions transitionSet = {};          /* initialization of  transition set*/
set-of-bins b = classifiedStates (allStates); /*generation of subsets of transformer methods*/
while  |pendingStates |> 0
        state r = extract (pendingStates);
        pendingState = pendingStates – {r};
        for each transformer class method  m
                /*generating transitions of the state r*/
                s = abstractInterpretationState (m, r);
                if s  ∉  allStates
                        pendingStates = pendingStates ∪ {s};
                        allStates = allStates  ∪ {s}
                endif
                transitionSet = transitionSet ∪ abstractInterpretationTransition (m,r,s)
        endfor
        /* updating subsets of transformer methods*/
        b = modifyBins (r, transitionSet, allStates);
        for each e ∈ b
                if s ∈ b {/*defining equivalence of states and merging equivalent  states*/
                        for each q  ∈ bin and s< > q
                                if equivalents (s, q) mergeStates (transitionSet, allStates, s, q) endif
                        endfor
                endif   endfor  endwhile
```

5 REVERSE ENGINEERING AT METAMODEL LEVEL

We specify reverse engineering processes as MOF-defined transformations. MOF allows capturing all the diversity of modeling standards and interchange constructs that are used in MDA. We call anti-refinement the process of extracting from a more detailed specification (or code) another one, more abstract, that is conformed by the more detailed one.

Figure 5 shows partially an ISM-C++ metamodel that includes constructs for representing classes, variables and functions. It also shows different kind of relationships between classes, for instance, a C++ class can have super classes or nested classes. Figure 6.a shows partially a PSM-C++ metamodel that includes constructs for representing classes, attributes, association ends and functions. The main difference between an ISM-C++ and a PSM-C++ is that the latter includes constructs for associations. The state diagram metamodel (Figure 6.b) defines a set of concepts than can be used for modeling discrete behavior through finite state transition systems such as state machines, states and transitions. We specify metamodel-based model transformations as OCL contracts that are described by means of a transformation name, parameters, preconditions, postconditions and additional operations. Transformation semantics is aligned with QVT, with the QVT Core in particular. In Figure 7 we partially exemplify a transformation from an ISM-C++ to a PSM-C++. This transformation uses both the specialized UML metamodel of C++ code and the UML metamodel of a C++ platform as source and target parameters respectively. The postconditions state relations at metamodel level between the elements of the source and target model. The transformation specification guarantees that for each class in C++ code there is a class in the PSM-C++, both of them with the same name, the same parent class, equivalent operations and so on. Besides, the PSM-C++ has a 'stateMachine' for each class having a significant dynamic behavior.

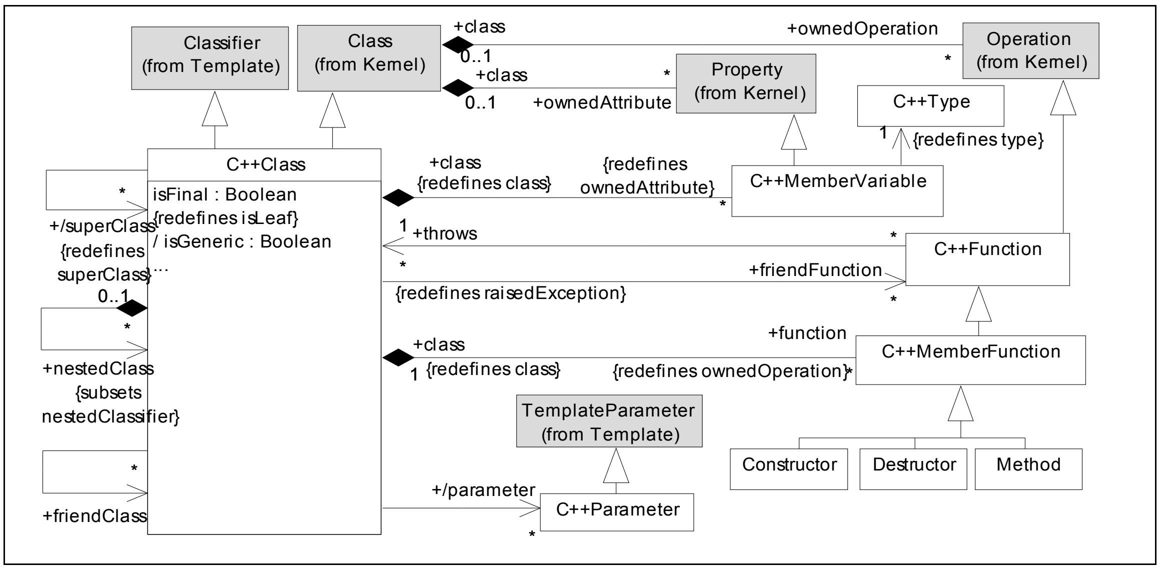
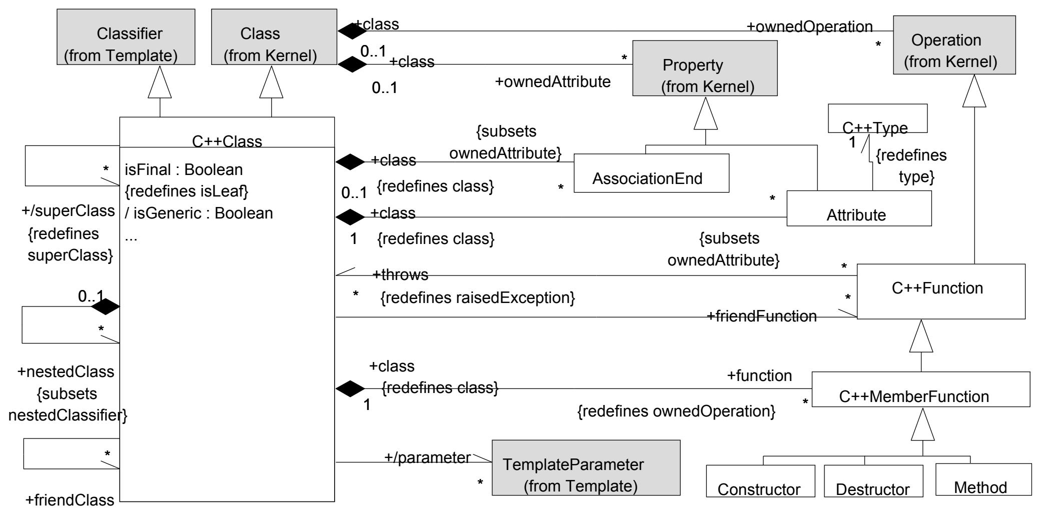
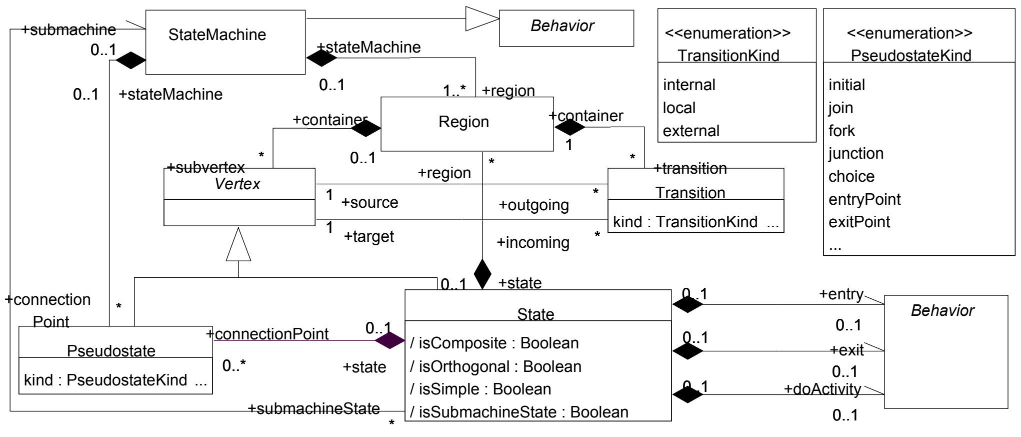


Figure 5. ISM C++ Metamodel



a. PSM C++ metamodel



b. PSM State Diagram metamodel

Figure 6. PSM Metamodel

6 TRANSFORMATION AT FORMAL LEVEL

This work is strongly integrated with previous ones that show how to formalize metamodels and metamodel-based transformations in terms of a formal metamodeling language called NEREUS. This language takes advantage of existing theoretical background on formal specifications and can be integrated with different algebraic languages.

NEREUS focused on interoperability of formal languages in MDD. A detailed description may be found at [10] that defines a bridge between MOF metamodels and NEREUS consisting of a system of transformation rules to convert automatically MOF into NEREUS. Also, the articles [10,11] shows how to integrate NEREUS with the Common Algebraic Specification Language (CASL) [2]. On the other hand, NEREUS allows specifying metamodels such as the Ecore metamodel, the specific metamodel for defining models in EMF (Eclipse Modeling Framework) [8]. Today, we are integrating NEREUS in EMF.

```
Transformation ISM-C++  to PSM-C++  {
parameter    sourceModel: ISM-C++Metamodel:: C++Project
             targetModel: PSM-C++Metamodel:: Package
postconditions
/* For each class 'sourceClass' in the sourceModel*/
sourceModel.ownedMember->select(oclIsTypeOf(C++Class))->forAll(sourceClass |
/*there is a class 'targetClass' in the targetModel so that both classes have the same name,*/
  targetModel.ownedMember-> select(oclIsTypeOf(C++Class))-> exists (
    targetClass | targetClass.name = sourceClass.name and
/*For each superClass  of 'sourceClass' there is a superClass  in 'targetClass' so that both super classes are
equivalent*/
    sourceClass.superClass-> forAll ( superCsource | targetClass.superClass-> exists (
        superCtarget |  superCtarget.classMatch(superCsource)  )    and
/*For each member function  of 'sourceClass' there is an operation in targetClass so that both operations are
equivalent*/
    sourceClass.C++MemberFunction->forAll(sourceF | targetClass.C++MemberFunction ->
    exists(targetF |    targetF.operationMatch(sourceF) ))  and
/*For each variable in 'sourceClass' whose type is a primitive or library type there is an attribute in
'targetClass' so that the attribute conform to the variable*/
    sourceClass.C++MemberVariable-> select (v |
        v.C++Type.oclIsTypeOf(Primitive) or v.C++Type.oclIsTypeOf(Library)  ) -> forAll
            (sourceVar | targetClass.attribute -> exists (  targetAtt |
                targetAtt.conformTo (sourceVar)  ) )  and
/*For each variable in 'sourceClass' whose type is a user defined type there is an association end in
'targetClass' so that the association end correspond to the variable:*/
    sourceClass.C++MemberVariable->select(v | v.C++Type.oclIsTypeOf(UserC++Class) )
        ->forAll     (sourceVar |  targetClass.associationEnd -> exists (  targetAssocEnd |
            targetAssocEnd.correspondTo (sourceVar)  )  )  and
/*If 'sourceClass' has some significant dynamic behavior, targetModel has  a 'stateMachine' so that:*/
    sourceClass.hasSignificantDynamicBehavior()  implies
    targetModel.ownedMember->select(oclIsTypeOf(StateMachine))-> exists (targetMachine |
/*'targetMachine' and 'sourceClass' have the same name and*/
        targetMachine.name = sourceClass.name   and
        /*For each modifier function in the 'sourceClass' there is a transition in 'targetMachine'*/
        sourceClass.C++MemberFunction-> select (f| f.isModifier() )-> forAll( f |
            targetMachine.region.transition-> exists( t | t.isCreatedFrom(f))) and … ) )
and …
}
```

Figure 7. ISM-C++ to PSM-C++ Transformation

7 CONCLUSIONS

MDA addresses the challenges of networked, changing system environment, providing a conceptual architecture that promotes portability, cross-platform interoperability, platform independent, better code quality and easier maintainability. Therefore, it is possible to define Domain-Specific languages adapted to new, industry-specific applications over diverse platforms.

This paper describes rigorous techniques to control the evolution of legacy systems in the context of science and engineering applications. We propose an MDA-based framework to reverse engineering code. Considering that testing, verification and consistency analysis are crucial activities in the modernization of systems that are critical to safety, security and economic profits, our approach stresses the integration of historic reverse engineering techniques with MDA and formal specifications.

We describe how to transform code into models in a higher level of abstraction that allows moving from these abstractions to new implementations.

REFERENCES

[1] Aho, A., Sethi, R. and Ullman, J. *Compilers: Principles, Techniques, and Tools* (Edition 2). Addison-Wesley (1985).

[2] Bidoit, M. and Mosses, P. *CASL User Manual- Introduction to Using the Common Algebraic Specification Language* (LNCS 2900). Heidelberg: Springer-Verlag (2004).

[3] Boronat, A., Carsi, J. and Ramos, I. Automatic reengineering in MDA using rewriting logic a transformation engine. *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR´05)* Los Alamitos: IEEE Computer Society (2005) 228-231.

[4] Canfora, G. and Di Penta, M. New Frontiers of reverse Engineering. Future of Software Engineering. *Proceedings of Future of Software Engineering.* FOSE 2007. Los Alamitos: IEEE Press (2007) 326-341.

[5] CASE TOOLS (2010). www.modelbed.net/mda_tools.html

[6] Daciuk, J. Incremental Construction of Finite-State Automata and Transducers, and their use in the Natural Language Processing. Ph. D. Thesis. Technical University of Gdansk (1998).

[7] Demeyer, S., Ducasse, S. and Nierstrasz, O. *Object oriented Reengineering Patterns*. Amsterdam:Morgan Kaufmann (2002).

[8] Eclipse: *The eclipse modeling framework* (2010) http://www.eclipse.org/emf/

[9] Fanta, R. and Rajlich, V. Reengineering of object oriented code. *Proceedings of the International Conference on Software Maintenance.* IEEE Press (1998) 238-246.

[10]Favre, L. A Formal Foundation for Metamodeling. Reliable Software Technologies- ADA Europe 2009. *LNCS* 5570. Springer-Verlag (2009) 177-191.

[11]Favre, L. *Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution*. Engineering Science Reference, USA (2010).

[12]MacDonald, A., Russell, D. and Atchison, B. Model driven Development within a Legacy System: An industry experience report. *Proceedings of the 2005 Australian Software Engineering Conference.* ASWEC 05. Los Alamitos: IEEE Press (2005) 14-22.

[13]Maoz, S. and Harel, D. On Tracing Reactive Systems. *Software and Systems Modeling* , Vol 9, Springer-Verlag (2010)

[14]MDA.*The Model Driven Architecture* (2005) www.omg.org/mda

[15]*MOF: Meta Object facility 2.0*. OMG Specification formal/2006-01-01 (2006).

[16]*OCL: Object Constraint Language. Version 2.2*. OMG: formal/2010-02-01 (2010).

[17]*QVT: MOF 2.0 Query, View, Transformation*. Formal/2008-04-03 (2008)

[18]Sommerville, I. *Software Engineering* (7th ed.). Addison Wesley (2004).

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

[20]*UML Infrastructure*. OMG Specification formal/ 2010-05-03 (2010).

[21]*UML Superstructure*. OMG Specification: formal/2010-05-05 (2010).