XRTSDIC: model transformation from PIM to PSM

G. Ramesh *

Lecturer In CSE, JNTUA College Of Engineering, Ananthapuramu, Andhara Pradesh, India-515002
*Corresponding author E-mail: ramesh680@gmail.com

Abstract

Computer Aided Software Engineering (CASE) has been growing faster in software industry. As part of it Model Driven Engineering (MDE) has been around for focusing on models and transforming them from one model to other model. The tool named Extensible Real Time Software Design Inconsistency Checker (XRTSDIC) proposed by us in previous paper supports UML modelling, design inconsistency checking and model transformation from UML to ERD to SQL. In this paper it is extended further to facilitate model transformation from PIM (UML class diagram) to PSM (source code). We proposed an algorithm and defined model transformation and consistency rules. The extended framework has provision for class relationship analysis and support for choosing different object oriented languages like C#, C++ and Java. While transforming the model, we used the concept of dialects. Dialect is the class with transformation functionality which has ability to adapt to syntax and semantics of chosen language. Different dialects are made available for different languages. Thus the proposed system is capable of transforming models and the prototype application we built and extended demonstrates the proof of concept. The empirical results revealed that the model transformation is consistent and accurate.

Keywords: Model Driven Engineering; Model Driven Engineering; Platform Independent Model (PIM); Platform Specific Model (PSM).

1. Introduction

Model transformation plays a vital role in the Computer Aided Software Engineering (CASE). Model transformation is the process of transforming a software design model from one form to another form. Generally there is transformation from platform independent model (UML model) to platform dependent model (classes of specific language). It is part of Model Driven Approach (MDA). In our previous paper we explored PIM to PSM that is from UML class diagram to SQL code. In this paper our focus is on the model transformation from PIM to PSM that is from UML class diagram to Java or C# or C++ classes. We used the notion of dialect to take care of syntax and semantics of chosen language while performing model transformation. We also used an algorithm for the same besides applying model checking and transformation using model consistency rules and transformation rules.

In the software engineering, especially for model transformation many researchers contributed. For instance Anshul et al. [1] proposed a method to convert UML class diagram to source code in the form of XML through XML transformation. Hama et al. [7] focused on automatic code generation. In the process they used class diagram and state chart diagrams besides using graph transformation technique. There are different approaches that are used to perform model transformation. It is evident in the researchers found in [9], [11], [17], and [21] to mention few.

In this paper we proposed a model transformation approach from PIM to PSM. We explored the model transformation from UML class diagram to source code for C++, Java and C#. We built different dialects to deal with language syntax and semantic differences. Our contributions in this paper are as follows.

• We built a prototype application that demonstrates the model transformation. The application supports Java, C# and C++ as target languages for PSM.
• We implemented the notion of dialect for handling syntax and semantic differences while performing model transformation.

The remainder of the paper is structured as follows. Section II provides review of literature. Section III presents the proposed system in detail. Section IV presents a case study. Section V provides experimental results while section VI concludes the paper.

2. Related works

This section provides review of literature on model transformation. Anshul et al. [1] proposed a method to convert UML class diagram to source code in the form of XML through XML transformation. Sawprakhon and Limpiyakorn [2] explored model transformation approach. Their approach generated sequence diagrams from the use case descriptions and class diagrams. They also used XML as an intermediate form while completing model transformation. Usman and Nadeem [3] extended the tool known as UIJECTOR in order to enhance it to support model transformation from UML diagrams to Java code. UML class, activity and sequence diagrams are given as input to the tool in order to generate Java source code. Sunitha and Samuel [4] MDA approach to convert a model from PIM to PSM. They performed model transformation from UML to Java code. They used XML as intermediate result before code is generated using Java language. Jilani et al. [5] focused on transforming DFD to UML diagrams by employing model transformation method. Arrassen et al. [6] explored model transformation known as Query View Transformation (QVT). They generated class diagrams from UML notations by using QVT and also Object Constraint Language (OCL).
Anshul et al. [1] proposed a method to convert UML class diagram to source code in the form of XML through XML transformation. Hama et al. [7] focused on automatic code generation. In the process they used class diagram and state chart diagrams besides using graph transformation technique. Pawar and Kulkarni [8] explored Atlas Transformation Language (ATL). Especially they worked on transforming UML design to a web based application using MVC2 architecture. Cambow [9] performed model transformation from UML class diagram to sequence diagram using transformation algorithm. Nikiforova et al. [10] studied the role of class diagram in UML in object oriented software development. As UML class diagrams play vital role in design and then implementation, their study assumed importance. Their study on class diagrams provided many conclusions that tell the usage of class diagrams in the software industry. Alakwaa and Salah [11] focused on the model transformation from ontology to another model known as content analysis model. Muhairat and Qudar [12] explored reverse engineering approach for converting a document containing forms into UML class diagrams and their results claimed high accuracy in the conversion process. Ovchinnikova and Asmina [13] explored different software tools available for obtaining UML sequence diagrams ad class diagrams from source code. In fact they enhanced TFM4MDA for achieving this. Cobet et al. [14] focused on the constraint programming using UML or OCL class diagrams. Achouri [15] explored the possible transformation from UML activity diagrams to EVENT-B model. The transformation is achieved by using institution theory. Esbai et al. [16] proposed an approach for model transformation from UML diagrams to N-tier web models. The model transformation language they used is MOF 2.0 QVT. Muttipiti and Padmaja [17] studied the problem of software maintenance. They worked out on the construction of software model graph and analyze the source code of the project. This is achieved by using Abstract Syntax Tree Method. Nikiforova [18] explored the model driven approach for transforming hemisphere to UML class diagram. Abdullah [19] presented a method for model transformation from UML class diagram to source code. Towards achieving this they used multi-threaded programming. Zhao [20] studied on the UML models and fault tree models. Then they proposed a model transformation approach from UML model to fault tree model. They made use of transformation language for achieving this. Rafe et al. [21] explored conversion of model from UML class diagram to relational table. Cherkhasin et al. [22] focused on a logical approach for OMG’s model transformation from UML class diagrams to set of classes. Magalhaes et al. [23] explored a platform independent approach for model transformation. Towards this end they proposed Model Transformation Profile (MTP) which is domain specific language used in the experiments. Gabmeyer et al. [24] studied software models and understood their dynamics. Then they proposed an approach for model-checking based verification in order to ensure that software models are consistent. Hassan et al. [25] explored the adaptation of OCL for MDE process. This is done after refactoring of their model. This approach is named as model refactoring.

In this paper we proposed a model transformation approach from PIM to PSM. We explored the model transformation from UML class diagram to source code for C++, Java and C#. We built different dialects to deal with language syntax and semantic differences.

3. Proposed approach

Easy Prior to discussing the proposed approach for model transformation from PIM to PSM, we introduce our framework proposed and extended earlier in [26], [27] and [28] respectively. The framework with extension (XML representation o API and generated source code API part) is shown in Figure 1. More details of it can be found in [26]. However, we provide an overview of it here as required. The purpose of the framework is to support modelling using UML notations, model consistency checking and automatic transformation from one model to another model. It has provision for choosing a modelling tool, consistency rule language and visualization technique using its personalized configuration. Thus the tool is flexible and personalized. The personalized settings are effective while drawing models. Further details on the framework are not in the scope of this paper. However, its extension (drawn in bold in Figure 1) is the focus of this paper.

![Fig. 1: Extended XRTSDIC.](image)
The model transformation from the UML class diagram to source code through an intermediate representation of model in the form of XML is the proposed extension to the framework. Stated differently, the work in this paper is confined to having a class diagram drawn in our tool and then perform transformation of the class diagram to different source codes based on the selection of language.

The class diagram is subjected to model checking and model transformation in order to generate source code automatically. It does mean that transformation rules and consistency rules are applied while performing automatic transformation. More details on the proposed work of this paper are shown in Figure 2.

Class diagram is the input to the framework. Once class diagram is drawn it can analyze the class diagram and make an XML representation of class diagram. The XML is the intermediate representation which is subjected to checking relationships. The relationships are analyzed as they can influence the code generation dynamics.

a) Relationship analysis

There are three important class relationships that exist in any class diagram. They are given importance in this paper as shown in Figure 3.

Class relationships are of three types namely IS A relationship which indicates inheritance between classes, HAS A relationship which indicates containment (one class object declared in another class) and USES or dependency relationship which indicates that a class can invoke business logic (BL) methods on another class. For all these relationships, the code generation will differ. That is the reason the relationship analysis is kept as part of the framework shown in Figure 2.

b) Role of dialect in model transformation

Since we are transforming UML class diagram to source code, there are many issues to be handled. As the source code needs to be generated automatically the underlying algorithm needs to know the semantics and syntaxes of the language. In other words, the algorithm needs to be trained in such a way that it can do its work correctly. Towards this end we used the concept of Dialect. The class hierarchy of dialect implementation is shown in Figure 4. The TransformationDialect is the class which has functionality for transforming from class diagram to source code based on the chosen language. However, it has basic abstract features and its sub classes have full knowledge on the language syntax and semantics. Therefore C#Dialect is the implementation of TransformationDialect. TransformationDialect is abstract in nature and cannot be used directly. In the same fashion JavaDialect and C++Dialect are used appropriately in order to have correct model transformation.

c) XRTSDDC approach for model transformation

Our XRTSDDC tool proposed in [26] follows the model transformation approach presented in Figure 5. The work in this chapter is an elaborated functionality of the model transformation section of the approach illustrated.
Class Diagram to Source code Transformation Rules
Class diagram contains class name, attributes and methods. The class diagram is transformed to corresponding source code (classes) according to the object oriented language selected.

Class Name \( \rightarrow \) Class Name
Consistency and transformation rule:
if a new class is created then the class name should be unique and should be available in class diagram.

Class Attribute \( \rightarrow \) Class Instance Variable
Consistency and transformation rule:
if a new attribute is created then the attribute name should be unique and should be available in the class attributes.

Class Attribute Type \( \rightarrow \) Class Attribute Type
Consistency and transformation rule:
if attribute type is determined then the attribute type should match or compatible with that of class attribute.

Class Method \( \rightarrow \) Class Method
This method should match or compatible with that of class method.

Class Method Arguments \( \rightarrow \) Class Method Arguments
The arguments in the generated classes should match arguments of method. However it is subject to the support in UML notation of class diagram.

Class Method Return Type \( \rightarrow \) Class Method Return Type
The return type of method should have same or compatible type in generated class.

Listing 1 – Consistency and transformation rules (Class Diagram \( \rightarrow \) Source code)

These rules are applied when the transformation takes place. Again the generated source code is based on the functionality of corresponding dialect chosen. The dialect can provide accurate source code generation.

Case study
A case study from the health care industry is considered and it is shown in figure 6. Our tool is used to draw class diagram as part of model design. Then the UML model is subjected to model consistency. As the class diagram is consistent it is used for model transformation as explained in this paper.

The class diagram contains different relationships. The inheritance relationship is between the following classes.

Person and Staff (Staff is inherited from Person)
Person and Patient (Patient is inherited from Person)
Person and Doctor (Doctor is inherited from Person)

The following are the class containing HAS A relationship in terms of aggregation.

Edit and Doctor (Doctor contained in the Edit class)
Edit and Staff (Staff contained in the Edit class)

Edit class is known as container class while the Doctor and Staff classes are named part classes.

The following are the classes that exhibit USES relationship in the UML model from healthcare case study.

Staff and Expenditure
Staff and TestOperators
Patient and Registration
Patient and Appointment
Patient and Reports
Registration and Ward

Generated source code with java dialect

```java
public class Person {
    private int StaffId;
    private String name;
    private String address;
    private String Sex;
    private int Salary;
    public void Create() {}
}

public class Patient extend person {
    private String Appdate;
    public void inPatient() {
    public void isPatient() {
}

public class Doctor extends person {
    private int Salary;
    public void drawSalary() {
}

public class Staff extends person {
    private int Salary;
    public void drawSalary() {
}

public class Edit {
    private int Id;
    private String name;
    private Staff s;
    private Doctor d;
    public void addDoctor() {
    public void addStaff() {
    public void delDoctor() {
    public void editDoctor() {
    public void editStaff() {
}

public class Ward {
    private int wardnumber;
    private int no of beds;
    private String name;
    public void bedStatus() {
    public void Name() {
}

public class Appointment {
    private date dt;
    private time tm;
    public void opAppt() {
    public void testAppt() {
}

public class Expenditure {
    private int amt;
    private int billnumber;
    public void giveSalary() {
}

public class Registration {
    private int patientId;
    public void Create() {
    public void Register() {
    public void Alloted() {
}

public class Reports {
    private int reportid;
    private String reportname;
    public void dispwardStatus() {
    public void dissadresStatus() {
    public void dispanInfo() {
}

public class Testoperators {
    private int patientId;
    private int id;
    private String flag;
    public void OpAppt() {
```
This way the prototype application we built is able to produce code in different languages using corresponding dialect. The source code generated for other languages is not provided here due to space constraint.

Duplicate code detection

After completion of model transformation, if there are inconsistencies with respect to code duplication that are to be notified. Towards this end, we improved our application to support duplicate code detection. The detection procedure is as follows.

Procedure: Duplicate Code Detection

Input: Source file
Output: Duplicate code detection results

01 Initialize source code vector SCV
02 Load source code into SCV
03 Initialize duplicate code vector DCV
04 Initialize colour vector CV
05 For each line in SCV
06 Compare current line with all other lines
07 If duplicates are found then
08 Add them to DCV
09 Add a random colour to CV
10 End If
11 End For
12 For each element of DCV
13 For each cv in CV
14 Associate cv to the element
15 Visualize duplicate code in the element
16 End For
17 End For

As shown in the above procedure, it is evident that the source code is subjected to an iterative process which helps in detecting duplicates and visualizing them. It is evident that the duplicate detection results are shown in figure 7. Each duplicate pair is shown in different colour. This provision in the prototype application can add value to the application to ensure that the transformed models are consistent.

4. Results

The proposed approach and the prototype application are evaluated with manual observation. The evaluation of the accuracy of code generation is performed by two senior software engineers invited from software industry.

Fig. 6: Class Diagram with IS A, HAS A and USES Relationships.
They studied the generated source code and opined that the application was working fine to transform model from PIM to PSM. Apart from the evaluation, the time taken for model transformation is recorded and presented here. The results reveal the differences in the performance among the dialects aforementioned in this paper.

<table>
<thead>
<tr>
<th>Model Elements</th>
<th>Execution Time (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.02</td>
</tr>
<tr>
<td>100</td>
<td>0.04</td>
</tr>
<tr>
<td>500</td>
<td>0.06</td>
</tr>
<tr>
<td>1000</td>
<td>0.08</td>
</tr>
<tr>
<td>1500</td>
<td>0.1</td>
</tr>
<tr>
<td>2000</td>
<td>0.2</td>
</tr>
</tbody>
</table>

As shown in Table 1, the execution time against number of model elements is presented. These results were recorded when JavaDialect is used.

As shown in Figure 8, the trend in the results revealed that the execution time is increased when model elements are increased.

<table>
<thead>
<tr>
<th># Model Elements</th>
<th>Execution Time (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.03</td>
</tr>
<tr>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>500</td>
<td>0.07</td>
</tr>
<tr>
<td>1000</td>
<td>0.09</td>
</tr>
<tr>
<td>1500</td>
<td>0.2</td>
</tr>
<tr>
<td>2000</td>
<td>0.3</td>
</tr>
</tbody>
</table>

As shown in Table 3, the execution time against number of model elements is presented. These results were recorded when C++Dialect is used.

<table>
<thead>
<tr>
<th># Model Elements</th>
<th>Execution Time (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.02</td>
</tr>
<tr>
<td>100</td>
<td>0.04</td>
</tr>
<tr>
<td>500</td>
<td>0.06</td>
</tr>
<tr>
<td>1000</td>
<td>0.08</td>
</tr>
<tr>
<td>1500</td>
<td>0.1</td>
</tr>
<tr>
<td>2000</td>
<td>0.2</td>
</tr>
</tbody>
</table>
As shown in Figure 8, the trend in the results revealed that the execution time is increased when model elements are increased.

5. Conclusions and future work

In this paper we studied model transformation from PIM (UML class diagram) to PSM (source code). In our previous works we proposed a tool with underlying framework known as Extensible Real Time Software Design Inconsistency Checker (XRTSDIC). This tool was supporting UML modelling, automatic design inconsistency checking and model transformation from UML to ERD to SQL. In this paper we extended the framework and tool in order to support model transformation from PIM to PSM. We proposed a framework in this paper for model transformation from UML class diagram to source code for different object oriented languages like C++, Java and C#. The transformation is subject to transformation rules and consistency rules. Besides the framework also supports different dialects for accurate transformation. As the languages different in the syntax and semantics we defined a Dialect API which takes care of transformation consistency in terms of syntax and semantic differences. A case study model from healthcare industry is considered to evaluate the proposed framework and the empirical results revealed that the framework is consistent and accurate in model transformation from UML class diagram to chosen object oriented language. This research can be extended further by implementing alternative model transformation techniques.

References


