Journal of Theoretical and Applied Information Technology
31st December 2017. Vol.95. No 24

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ECLIPSE JDT-BASED METHOD FOR DYNAMIC ANALYSIS INTEGRATION IN JAVA CODE GENERATION PROCESS

[1]A. ELMOUNADI, [2]N. BERBICHE, [3]F.GUEROUATE, [4]N. SEFIANI
System Analysis, Information Treatment and Industrial Management Laboratory,
Mohammed V University in Rabat, Morocco.
E-mail : [1]a.elmounadi@gmail.com, [2]nberbiche@hotmail.com, [3]guerouate@gmail.com, [4]nasefiani@gmail.com

ABSTRACT

In software engineering, The Unified Modeling Language (UML) is generally used as the de facto standard notation for modeling in the analysis and the design of the object oriented software systems. As known, throughout the modeling phase, the structural and behavioral elements go together, because they have complementary relationship in the understanding of systems architecture. However, structural analysis has always attracted the interest of designers more than the behavioral analysis, due to its prominent role in the code generation processes. This vision influenced the computer-aided software engineering (CASE) tools and the model-driven engineering (MDE) approach. As a result, by using CASE tools and taking up MDE approach as it is, the obtained code artefacts are incomplete and become the developers responsibility. Therefore, the model's abstraction is broken, which leads to a paradoxical situation while adopting model-driven development. To cope with this challenge, the purpose of our paper is to bring balance to the design stage by integrating the behavioral analysis into the code generation processes, in order to empower and promote delivering applications without the need for hand coding.

Keywords: *MDE, UML, Dynamic Analysis, Abstract Syntax Tree, Code Generation.*

1. INTRODUCTION

The latest versions of the Object Management Group (OMG) [1] standards provide well-established notations to the platforms specification of structural and behavioral design of software. Thereby, in UML[2], graphical notations have become good enough for a detailed modeling and a simplified human communication. For example, the activity diagram meta-model [2] actually proposes all the elements needed for describing, at the smallest detail, the body of methods in a software architecture. In parallel, the modeling tools vendors were in the obligation to follow this evolution in alignment with the newest possibilities proposed by the specifications. Therefore, models become productive elements instead of being contemplative, as they start to participate in the development lifecycle thanks to the rise of Model-driven engineering [3], which focuses on direct code generation from models. Thus, in model-driven engineering context, models incur a number of operations in order to produce executable source code, this process is commonly called the Model-to-text transformation. Many approaches allow achieving the model transformations. However, these approaches stay not suitable to handle the behavioral modeling aspect even if they allowed a considerable advance besides structural modeling. Therefore, the transformation processes based on these approaches remain incomplete, which requires the intervention of developers teams. The described situation is out of keeping with the Model-driven engineering approach that calls to protect the model's abstraction. The aim of this paper is twofold: first, it introduces a new model transformation method based on abstract syntax tree [4], which allows an end-to-end integration of the activity diagram in the model transformation process. This will help to handle the behavioral modeling aspect and perform a full-featured code generation process from UML models. Then, the paper presents the implementation of this method and simulates a concrete case study. The rest of this paper is organized as follows: The second section presents model-driven engineering and models transformation as the research context. Then, multiple previous researches are highlighted in the background part. Section 3 presents the assimilated methodology in the current study. Section 4 describes the Eclipse JDT-based transformation method and discusses the functioning of the given

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

method through the implementation part consolidated by the experimental validation, before moving to the conclusion and the future works.

2. RESEARCH BACKGROUND

2.1 Model-driven engineering

When model-centric development [5] meets the software engineering concepts, we obtain the model-driven engineering approach. In this approach, models play a prominent role, reducing by this fact the complexity of software development process and promoting communication among the several stakeholders. This implies a remarkable gain in productivity by maximizing compatibility between systems. Model-driven engineering assumes that models are sustainable over time, while development technologies are constantly changing. Thus, models become productive elements and become the primary artifacts that drive the whole development process also, in opposite to the code-centric approach, known by limiting the models to a descriptive role only. Several MDE initiatives exist, like OMG's Model-Driven Architecture (MDA) [6] and Microsoft software Factories [7].

The model transformations are the key concept in model-driven engineering. They constitute the most important operations applied to models in order to automate creation of target models from source models.

Model transformations have been classified in many ways [8]–[10]. In general-purpose, two kinds of model transformations exist:

- ✓ Model-to-model transformations (M2Mt),
- ✓ Model-to-text transformations (M2Tt).

Figure 1: Simplified MDE transformation process

In one hand, a model-to-model transformation is generally horizontal (i.e. acts at the same abstraction level) and it can be either endogenous or exogenous, according to the corresponding meta-models of the source and target model. In fact, a model transformation is endogenous when the same meta-model defines both the source and the target model. Whereas, a model transformation is exogenous when the source and the target model complies with two distinct meta-models. This kind of model transformation usually involves a refinement or customization to an execution platform. On the other hand, a model-to-text transformation represents code generation from the model. It consists to translate the input model into a concrete syntax, thereby producing code artifacts ready to compilation and execution.

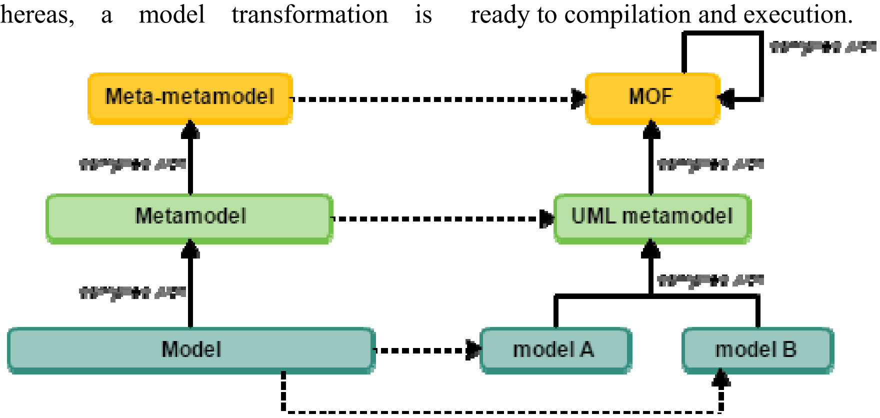


Figure 2: Example of the OMG stack meta-model compliance

The next section highlights several previous researches in the same context, with a view to bring out the multiple motivations that lead us to this work.

2.2 Motivations

The main purpose of the current study is to allow an end-to-end activity diagram integration in the code generation process. Activity diagram will represent the behavioral aspect in the current study. While adopting model-driven development in a

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

software engineering process, the use of a Computer Aided Software Engineering (CASE) tool is unavoidable. Thus, the first step was to check either if these tools allow code generation from behavioral diagrams realized during the modeling phase. CASE Tools that have been tested for this purpose are: MagicDraw [11], IBM Rhapsody [12], Enterprise-Architect [13], Objecteering [14], Modelio [15], Papyrus [16], Bouml [17] and UMLDesigner [18]. However, after testing these CASE tools, we found that they all integrate only the class diagram in their code generation process, whereas the other diagrams remain unexploited. As known, class diagram is the eponymous structural diagram in UML.

Table 1: Tested modeling tools

Name	License	Diagrams used in the code generation process
MagicDraw	Commercial	Class diagram
IBM Rhapsody	Commercial	Class diagram
Enterprise-Architect	Commercial	Class diagram
Objecteering	Commercial	Class diagram
Modelio	General Public License / Commercial	Class diagram
Papyrus	Eclipse Public License	Class diagram
BOUML	GPL / Commercial	Class diagram
UMLDesigner	EPL	Class diagram

Commercial tools *(MagicDraw, IBM Rhapsody, Enterprise-Architect and BOUML)* offer the possibility to incorporate method definitions via the modeler itself. Independent changes to the model and code can be merged without destroying data in both code and model. Once again, such a technique causes the model's abstraction breaking, due to the interweaving between modeling operations and coding lifecycle, which goes against the Model-driven engineering philosophy.

Therefore, in order to overcome this situation, it was necessary to understand the model transformation concepts. In previous works [10], [19]–[21], the authors have presented the classifications of the model transformation approaches. They also presented some of the suited tools and languages dedicated for these transformation approaches. Figure 3 gives a summary of this classification. By analyzing the Figure 3, it is obvious that M2T transformations area is still failing to gain the interest of researchers and tools vendors. In fact, template method is the exclusive approach used for executing this kind of transformations. However, this approach entails several disadvantages: it is somewhat error prone because the target source-code file is treated as a flat-file. Therefore, the manufacturer of the transformation must have expertise regarding the target language; he must also be a modeler and developer at the same time. Nevertheless, due to the amount of work that has to be done, sometimes error can skip into the heap. In addition, the transformation patterns could become obsolete if they do not follow the evolution of the target language versions, which causes another downside of this approach: the lack of scalability.

At the same time, several works [22]–[25] have been made to get the activity diagram transformation in the model transformation process. The most interesting idea was presented in [23], the authors considered the source programming language (Java in this case) as a model too. A meta-model for the Java language was provided, and the transformation rules were written in Atlas transformation language (ATL) [26]. Indeed, the work was clever by adopting this technic. However, the idea of treating a programming language as a model was not good enough. In fact, the best way of representing a program loaded in memory is the abstract syntax tree. Therefore, we gathered all the necessary elements to warrant the proposition of a new M2T transformation approach that could rely on the issues discussed above. The next section presents the methodology and basic concepts of the proposed JDT-based transformation method.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

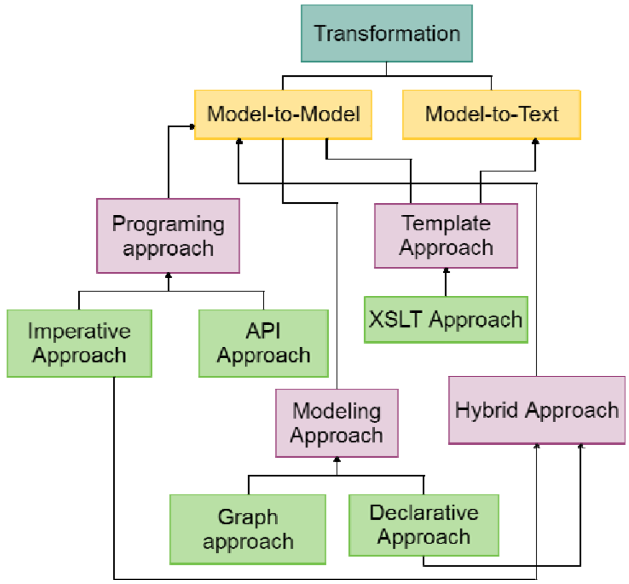


Figure. 3: Classification of the model transformation approaches

3. METHODOLOGY

3.1. UML2 Activity Diagram

Activity diagram is one of the behavioral diagrams proposed by UML. Originally intended for workflow description, it has evolved to allow also algorithmic translation of use cases, providing a microscopic view of the system. Therefore, an activity diagram can graphically represent the behavior of a method, and this is the most interesting reason to choose it as the input diagram of our model transformation process.

There are seven levels for activity diagrams representation [27]:

- ✓ Fundamental: The fundamental level defines activities as containing nodes and edges.
- ✓ Basic: This level includes data flow between actions, and includes InitialNode and ActivityFinalNode.
- ✓ Intermediate: The intermediate level supports fundamental and basic levels, and includes decision nodes.
- ✓ Complete: The complete level supports edge weights.
- ✓ Structured: supports sequences and loops.
- ✓ CompleteStructured: adds support for data flow output pins of sequences, conditionals, and loops.
- ✓ Extra-Structured: includes exception handling as found in object-oriented programming languages.

The Figure 4 presents a part of an adapted version of activity diagram meta-model for Java programming from completeStructured level.

3.2. Abstract Syntax Tree:

Compilation is the set steps for translating human readable source code, to executable binary code intended to run on a computer processor [4].

The compilation process takes place in three key steps:

- ✓ Lexical analysis: this stage is about scanning the source code to identify symbols that represent identifiers, constants, variables, language keywords and eliminate unnecessary elements considered as comments and line breaks, etc.
- ✓ Syntax analysis: also called grammatical analysis, it constitutes the parsing phase. The parser handles the tokens produced during the lexical analysis phase and must verify that it can be generated by the grammar. In grammar, two types of symbols can be distinguished: terminal and non-terminal. Terminal symbols are the language keywords. The non-terminal symbols represent variables, constants and functions created by the developer. At this level, the parser attempts to build an in-memory structure representation. This structure is called abstract syntax tree (AST). An AST is a tree representation of data structure of a program. It consists of a set of instances from abstract syntax language elements.

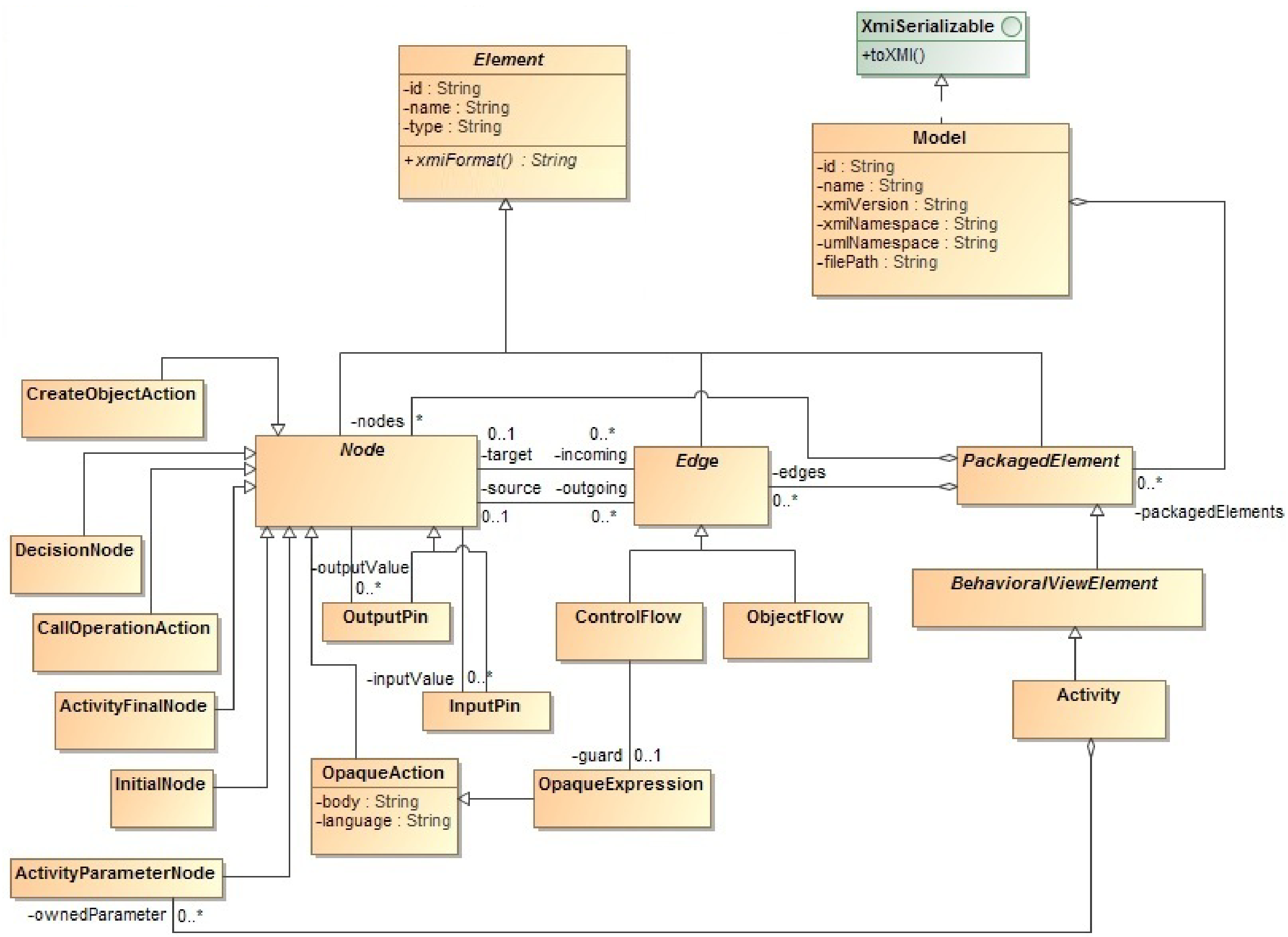


Figure: 4: Activity diagram meta-model adapted to Java programming, completeStructured level.

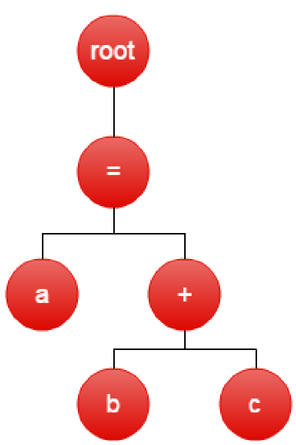


Figure 5: Example of an Abstract Syntax Tree

✓ Semantic analysis: at this stage, the compiler inspects if the program is written in a logical way. For example, it is inappropriate to use a variable before its declaration or try to affect a string value to an integer variable.

3.3. Java Development Tools

Java development tools (JDT) [28] is an integrated plugin to the Eclipse platform that allows managing Java projects. Syntax coloration, syntax error detection and project overview are all full-featured Java IDE added to the Eclipse platform with this plugin.

The project is organized into five main packages:

✓ JDT-APT: for JDT Annotation Processing Tool, it provides the capability to recognize and process annotations. Annotations appeared for the first time in Java 5.

✓ JDT-Debug: implements Java debugging support.

✓ JDT-Text: manages the text editing into the IDE. It facilitates the text manipulation and offers support for text formatting, auto-completion, hover help, rule based styling and more.

✓ JDT-UI: represents the implementation of Java IDE user interface.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

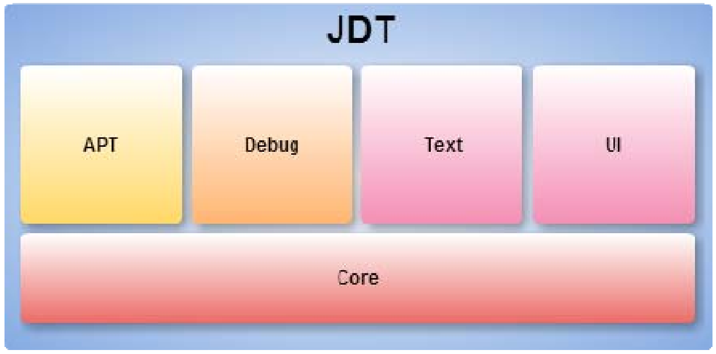


Figure 6: JDT packages organization

✓ JDT-Core: The package JDT-Core contains a set of classes, which represent an API for manipulating the source code of a Java file as a structured document. The Java file is loaded into memory as an Abstract Syntax Tree (AST). An AST is the abstract representation of the source code structure as a tree. JDT-Core contains a sub-package called DOM/AST. It contains all the classes that represent the Java meta-model, where each element of the abstract syntax tree instantiates a given class. The Figure 7 below shows a class diagram that represents a part of the meta-model class hierarchy.

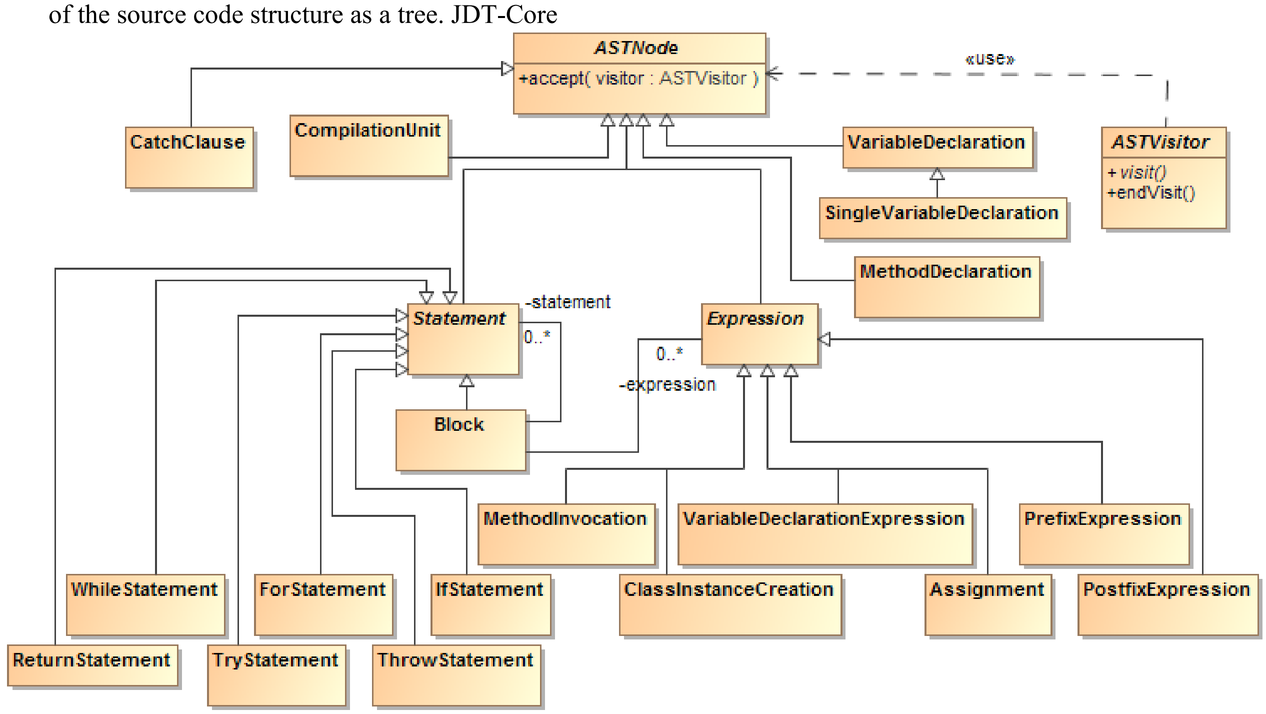


Figure 7: A part of Java meta-model

Table 2: A part of the Java meta-model elements description

	Name	Description
1	ASTNode	Abstract superclass of all AST nodes
2	ASTVisitor	Abstract visitor of each node in the AST
3	CompilationUnit	Representation of the Java file as a compilation unit
4	MethodDeclaration	Method declaration AST node type
5	VariableDeclaration	Variable declaration concept, could handle multiple variable declarations.
6	SingleVariableDeclaration	A single variable declaration, allow specifying method parameters also.
7	Expression	Notion of abstract expression
8	Statement	Notion of abstract statement, the smallest standalone element of Java language
9	Block	Represent code block type
10	MethodInvocation	Represent method calling
11	VariableDeclarationExpression	Variable declaration expression which consists of variable declaration fragments
12	ClassInstanceCreation	Creation of an object with the 'new' operator
13	Assignment	An Expression based on '=' operator with 2 hand sides
14	PrefixExpression	An expression prefixed by an operator, generally increment or decrement operator
15	PostfixExpression	An expression post fixed by an operator, generally increment or decrement operator
16	IfStatement	Represent the if-then-else statement
17	ForStatement	Represent the for loop statement
18	WhileStatement	Represent the while loop statement
19	TryStatement	These two elements go generally together to represent the try-catch statement in exceptions processing
20	CatchClause	
21	ThrowStatement	Rising exception statement
22	ReturnStatement	Return statement in a method body

The table above gives the technical description of each element in the Java meta-model [29].

4. EXPERIMENTAL VALIDATION

Now, to give you an insight about the interest of using the JDT-based transformation method, the following section will show how to accept an Activity Diagram as an input model to produce the corresponding Java source code as target text, thus demonstrating the benefit of the chosen method. Indeed, this approach can work with any other programming language.

4.1. Implementation

The JDT-based transformation method is intended to realize model-to-text transformations. This method is based on the abstract syntax tree's concept. Unlike the template approach, the proposed method is less error prone and offers more scalability. This method uses the JDT API described above, which allows to manipulate the internal structure of a Java program. However, this method is based on an AST-based transformation approach, which can be considered as a new approach. In this context, JDT API will be used to build the code structure. The Figure 8 illustrates the positioning of the new approach, vis-à-vis the different existing model transformation approaches.

As known, UML meta-model is conformed to Meta-Object Facility (MOF) [30] standard. In the same context, OMG proposes a serialization standard for serializing MOF objects called XML Metadata Interchange (XMI) [31]. This serialization format offers an XML representation of the diagram elements. The Figure 10 below shows an activity diagram and the corresponding XMI representation of each graphical notation in the diagram. The JDT-based transformation method operates on the XMI file as an input file in order to extract the model elements then to perform the suitable transformation for each element.

Step 1: The XMI file parsing

The input XMI file undergoes a parsing step in order to perform the diagram nodes selection. Then, we proceed to the diagram nodes browsing by invoking the visitor design pattern [32]. The implementation of such a mechanism required introducing some adjustments on the activity diagram meta-model. Figure 11 below illustrates a part of the modified activity diagram meta-model.

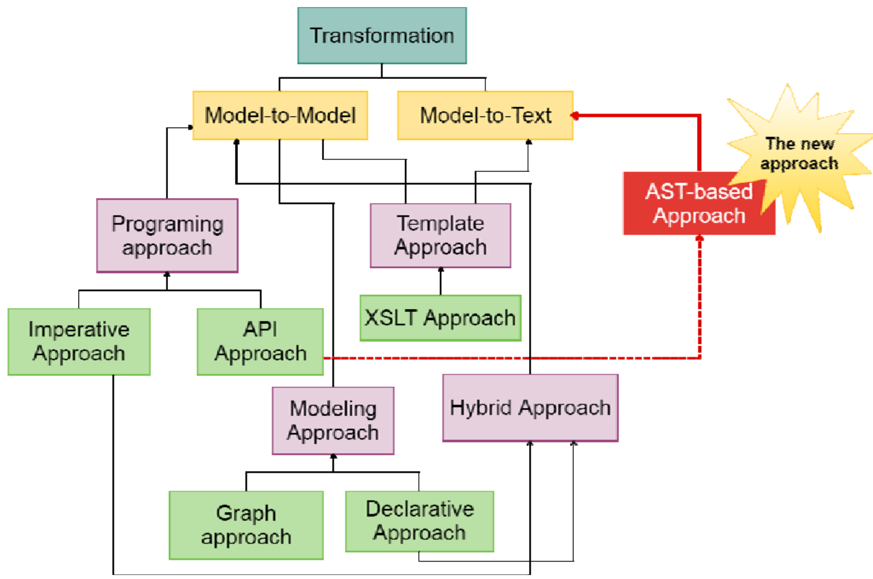


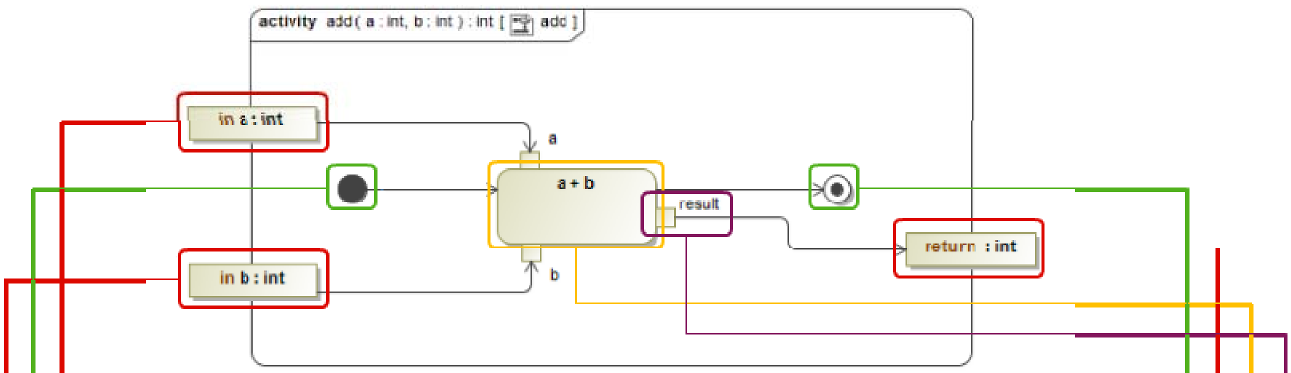
Figure 8: Positioning of the AST-based transformation approach.

Figure 9: The Eclipse JDT-based code generator tool architecture.

<u>Step 2</u>: From activity diagram to AST conversion

Afterwards, the visiting mechanism is coupled to the using of the JDT API. The construction of the AST, corresponding to the visited nodes of the activity diagram given as input, is among the objectives to be achieved. The following table 3 shows some of the activity diagram meta-model's elements and their corresponding Java language elements. We can observe that an UML element is not necessary represented by a Java element *(InitialNode, ActivityFinalNode)*, and vice-versa *(Assignment, PostfixExpression…)*. Thereby, the M2T transformation concerning UML Activity diagram to Java language is not bijective. Once again, the template approach will not be suitable to perform such a case, because it will be difficult for it to handle the transformation with this multitude of scattered elements. The JDT-based transformation method shows its efficiency by handling the same cases to obtain rigorous results. The main class in our implementation is XMIVisitor class. This abstract class defines the visit methods as advocated in the visitor design pattern [32] for all XMI elements related to the UML standard.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195




```xml
<packagedElement xmi:type="uml:Activity" xmi:id="_17_0_5_1a700d7_1467061360403_974140_3693" name=
"add" isReentrant="true" specification="_17_0_5_1a700d7_1467061218862_958390_3423">
  <ownedParameter xmi:id="_17_0_5_1a700d7_1467061539706_694267_3785" name="a" visibility="public">
  <ownedParameter xmi:id="_17_0_5_1a700d7_1467061556966_86267_3801" name="b" visibility="public">
  <ownedParameter xmi:id="_17_0_5_1a700d7_1467062599830_69520_3988" name="" visibility="public"
  direction="return">
  <node xmi:type="uml:ActivityParameterNode" xmi:id="_17_0_5_1a700d7_1467061539704_275747_3784"
  name="a" visibility="public" outgoing="_17_0_5_1a700d7_1467063338102_310833_4110" parameter=
  "_17_0_5_1a700d7_1467061539706_694267_3785">
  <node xmi:type="uml:ActivityParameterNode" xmi:id="_17_0_5_1a700d7_1467061556965_264219_3000"
  name="b" visibility="public" outgoing="_17_0_5_1a700d7_1467063343128_440616_4115" parameter=
  "_17_0_5_1a700d7_1467061556966_86267_3801">
  <node xmi:type="uml:ActivityParameterNode" xmi:id="_17_0_5_1a700d7_1467062599829_431059_3987"
  name="return" visibility="public" incoming="_17_0_5_1a700d7_1467063375523_175106_4120" parameter
  ="_17_0_5_1a700d7_1467062599830_69520_3988">
  <node xmi:type="uml:OpaqueAction" xmi:id="_17_0_5_1a700d7_1467062766580_890877_4008" name=""
  visibility="public" outgoing="_18_5_2070117_1490028533695_239069_4994" incoming=
  "_18_5_2070117_1490028493796_918693_4981">
    <inputValue xmi:id="_17_0_5_1a700d7_1467063214094_883180_4082" name="b" visibility="public"
    incoming="_17_0_5_1a700d7_1467063343128_440616_4115">
      <type xmi:type="uml:PrimitiveType" href=
      'pathmap://UML_LIBRARIES/JavaPrimitiveTypes.library.uml#int'/>
    </inputValue>
    <inputValue xmi:id="_17_0_5_1a700d7_1467063235593_557332_4095" name="a" visibility="public"
    incoming="_17_0_5_1a700d7_1467063338102_310833_4110">
    <outputValue xmi:id="_17_0_5_1a700d7_1467062883002_516117_4047" name="result" visibility=
    "public" outgoing="_17_0_5_1a700d7_1467063375523_175106_4120">
    <body>a + b</body>
  </node>
  <node xmi:type="uml:InitialNode" xmi:id="_18_5_2070117_1490028483360_828521_4974" name=""
  visibility="public" outgoing="_18_5_2070117_1490028493796_918693_4981"/>
  <node xmi:type="uml:ActivityFinalNode" xmi:id="_18_5_2070117_1490028527670_456487_4986" name=""
  visibility="public" incoming="_18_5_2070117_1490028533695_239069_4994"/>
  <edge xmi:type="uml:ObjectFlow" xmi:id="_17_0_5_1a700d7_1467063338102_310833_4110" name=""
  visibility="public" source="_17_0_5_1a700d7_1467061539704_275747_3784" target=
  "_17_0_5_1a700d7_1467063235593_557332_4095">
  <edge xmi:type="uml:ObjectFlow" xmi:id="_17_0_5_1a700d7_1467063343128_440616_4115" name=""
  visibility="public" source="_17_0_5_1a700d7_1467061556965_264219_3800" target=
  "_17_0_5_1a700d7_1467063214094_883180_4082">
  <edge xmi:type="uml:ObjectFlow" xmi:id="_17_0_5_1a700d7_1467063375523_175106_4120" name=""
  visibility="public" source="_17_0_5_1a700d7_1467062883002_516117_4047" target=
  "_17_0_5_1a700d7_1467062599829_431059_3987">
  <edge xmi:type="uml:ControlFlow" xmi:id="_18_5_2070117_1490028493796_918693_4981" name=""
  visibility="public" source="_18_5_2070117_1490028483360_828521_4974" target=
  "_17_0_5_1a700d7_1467062766580_890877_4008">
  <edge xmi:type="uml:ControlFlow" xmi:id="_18_5_2070117_1490028533695_239069_4994" name=""
  visibility="public" source="_17_0_5_1a700d7_1467062766580_890877_4008" target=
  "_18_5_2070117_1490028527670_456487_4986">
</packagedElement>
```

Figure 10: UML graphical notations vs XMI serialization example

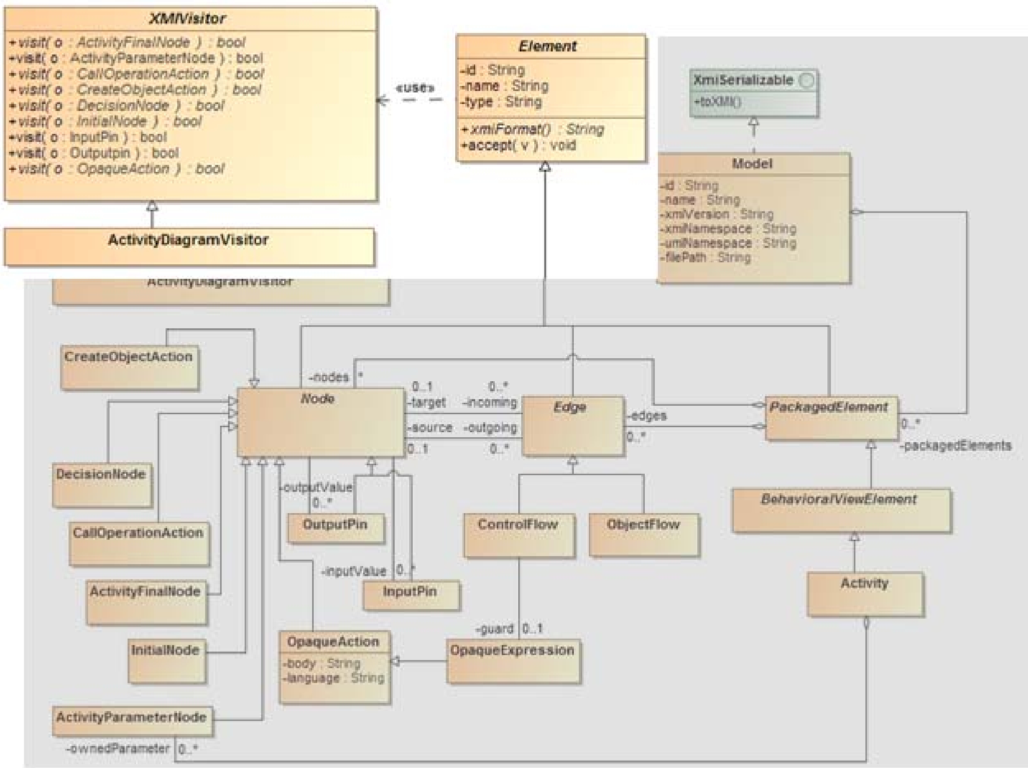


Figure 11: The modified Activity diagram meta-model

Table 3: Basic UML activity diagram actions & Java language elements matching

	UML Activity diagram elements	Java language elements	General concept
1	InitialNode	-	Method beginning
2	ActivityFinalNode	-	Method ending
3	CallOperationAction	MethodInvocation	Method calling
4	CreateObjectAction	ClassInstanceCreation	Object instantiation
5	AddVariableValueAction	VariableDeclarationExpression	Variable declaration
6	ConditionalNode/DecisionNode	IfStatement	If-then-else statement
7	-	Assignment	Assigning a value to a variable or an object
8	Activity/ Composite Activity	Block	Block statement
9	-	PostfixExpression	Increment or decrement a variable
10	-	PrefixExpression	Increment or decrement a variable
11	LoopNode	WhileStatement	Loop statement
		ForStatement	
12	InputPin	SingleVariableDeclaration	Temporary variable declaration
13	OutPutPin		
14	ActivityParameterNode	SingleVariableDeclaration	Method parameters
		ReturnStatement	Return statement when the direction's parameter is return

We can observe that an UML element is not necessary represented by a Java element *(InitialNode, ActivityFinalNode)*, and vice-versa *(Assignment, PostfixExpression…)*. Thereby, the M2T transformation concerning UML Activity diagram to Java language is not bijective. Once again, the template approach will not be suitable to perform such a case, because it will be difficult for it to handle the transformation with this multitude of scattered elements. The JDT-based transformation method shows its efficiency by handling the same cases to obtain rigorous results.

Journal of Theoretical and Applied Information Technology
31st December 2017. Vol.95. No 24

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The main class in our implementation is XMIVisitor class. This abstract class defines the visit methods as advocated in the visitor design pattern [32] for all XMI elements related to the UML standard. The ActivityDiagramVisitor class redefines the XMIVisitor visit methods related to the activity diagram elements only, as described in the UML standard too. By the way, this class contains five main properties:

- ✓ A compilation unit: this property represents the compilation unit that will be useful for the serialization of the generated code.
- ✓ An AST: this property represents the abstract syntax tree that holds the code instructions.
- ✓ A type declaration: this field represents the wrapping class of the activity/method.
- ✓ A method declaration: this property defines the method declaration and prepares the diagram transformation.
- ✓ A block: the block of instructions contained between { }.

```
public class ActivityDiagramVisitor extends XMIVisitor {

    private CompilationUnit unit;
    private AST astUnit;
    private TypeDeclaration class_;
    private MethodDeclaration method;
    private Block block;
```

The activity node visit means the creation of a new method. The following code snippet shows how to create a new method.

```
public boolean visit(Activity o) {

    method = astUnit.newMethodDeclaration();
    method.setConstructor(false);

    method.modifiers().add(astUnit.newModifier(
            Modifier.ModifierKeyword.PUBLIC_KEYWORD));
    method.setName(astUnit.newSimpleName(o.getName()));

    method.setReturnType2(
            astUnit.newPrimitiveType(PrimitiveType.VOID));

    for (int i = 0; i < o.getNodes().size(); i++) {
        o.getNodes().get(i).accept(this);
    }

    return false;
}
```

The child nodes of the activity accept the visitor. For example, the visiting of an InitialNode means the beginning of the method definition that will be contained in a block instruction.

```
public boolean visit(InitialNode o) {
    block = astUnit.newBlock();

    return false;
}
```

With the same logic, the visiting of an ActivityFinalNode constitutes the method ending.

```
public boolean visit(ActivityFinalNode o) {
    method.setBody(block);

    return false;
}
```

During the nodes visiting step, when the visitor encounters a DecisionNode (which generally represents an if-then-else statement in programming languages), the visitor behaves as follows:

```
public boolean visit(DecisionNode o) {

    AST ast = block.getAST();

    IfStatement ifStatement = ast.newIfStatement();

    String condition = o.getCondition().getBody();
    ASTParser subparser = ASTParser.newParser(AST.JLS3);
    subparser.setKind(ASTParser.K_EXPRESSION);
    subparser.setSource(condition.toCharArray());

    InfixExpression expression = (InfixExpression)
            subparser.createAST(null);
    InfixExpression i = (InfixExpression)
            ASTNode.copySubtree(astUnit, expression);
    i.delete();
    ifStatement.setExpression(i);

    block.statements().add(ifStatement);

    return false;
}
```

At the end of the activity node visiting, the following actions are performed.

```
public boolean endVisit(Activity o) {
    class_.bodyDeclarations().add(method);
    unit.types().add(class_);

    return false;
}
```

Concerning the variable typing, we established a table mapping between UML typing and Java typing related to the primitive types. The table 4 below shows the types mapping:

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 4: Table mapping between UML primitive types and Java primitive types

UML primitive type	Java primitive type
pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#Boolean	boolean
pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#Integer	int
pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#Real	float
pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#String	String
pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#UnlimitedNatural	int

```xml
<result xmi:id="_18_5_2070117_1490380928210_535993_5122" name="result" visibility="public">
  <type xmi:type="uml:PrimitiveType"
  href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#Integer"/>
</result>
```

Figure 12: Serialization of Node type in XMI

<u>Step 3</u>: The AST serialization

Finally, the concrete syntax is obtained from the AST built previously through the following serialization mechanism.

```java
TypeDeclaration _class = (TypeDeclaration) unit.types().get(0);
try {
    PrintWriter p = new PrintWriter(new File(_class.getName() + ".java"));
    p.write(unit.toString());
    p.close();

} catch (FileNotFoundException ex) {
    ex.printStackTrace();
}
```

4.2. Experimental results

We implemented a series of tests in order to provide scientific proof to the good functioning of the given approach. The following activity diagram represents an arithmetic method that returns the addition of two integers. The obtained source code in Java language after the model transformation is described below. Therefore, we can notice that the obtained results are congruent.

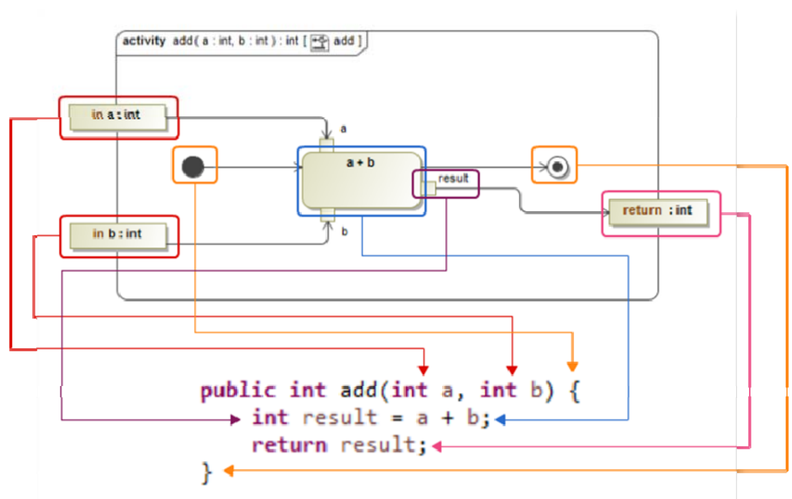


Figure 13: Example of JDT-based transformation method application

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

5. CONCLUSION AND FUTURE WORKS

This paper has introduced the new JDT-based transformation method as a new AST-based transformation approach in model-to-text transformations, which is intended to bridge the gap between the behavioral diagrams in UML modeling and Java code generation. This method comes to compete with the concept of executable UML models [33]–[36], which require more maturity. It is now possible to graphically represent a method-body of a given class and generate the corresponding source code like in visual programming [37], but in a more professional context. Among all UML diagrams available, the choice fell on the activity diagram because it is best-suited one to represent code instructions. Nevertheless, some points require more attention; the method must allow managing higher levels of activity diagram meta-model to reach advanced coding levels. It must also ensure generation for multi-threading and exception handling. The aim of the future work is to bring together the work presented in [38] and the work presented in this paper with improvements in order to provide a new software engineering tool that will allow a full round-trip engineering related to Java technologies in a Model-driven software engineering context.

REFRENCES:

[1] Object Management Group, “http://www.omg.org/,” *OMG TM*, 10-Mar-2016. [Online]. Available: http://www.omg.org/. [Accessed: 22-Jan-2017].

[2] Object Management Group, “OMG Unified Modeling Language TM (OMG UML) Version 2.5,” *Object Manag. Group Pct07-08-04*, 2015.

[3] Douglas C. schmidt, “Model Driven Engineering,” IEEE computer society, 2006.

[4] A. JavadiAbhari et al., “ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs,” in Proceedings of the 11th ACM Conference on Computing Frontiers, New York, NY, USA, 2014, pp. 1:1–1:10.

[5] A. Forward and T. C. Lethbridge, “Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals,” in Proceedings of the 2008 international workshop on Models in software engineering, 2008, pp. 27–32.

[6] Nawfal El Moukhi, Ikram El Azami, and Aziz Mouloudi, “Towards a new method for designing multidimensional models (in press),” International Journal of Business Information Systems.

[7] J. Greenfield and K. Short, “Software factories: assembling applications with patterns, models, frameworks and tools,” in Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2003, pp. 16–27.

[8] C. K. and H. S., “Feature-based survey of model transformation approaches.,” IBM Systems Journal 45 (3), pp. 621–645, 2006.

[9] P. Stevens, “A landscape of bidirectional model transformations.,” Generative and transformational techniques in software engineering II, Springer, pp. 408–424.

[10] L. Tratt and M. Gogolla, Eds., Theory and practice of model transformations: third international conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010: proceedings. Berlin ; New York: Springer, 2010.

[11] N. Magic, Inc., MagicDraw: Architecture Made Simple. 2011.

[12] IBM, “IBM Rhapsody.” [Online]. Available: http://www-03.ibm.com/software/products/en/ratirhapfami. [Accessed: 25-Mar-2017].

[13] Sparx systems, “Enterprise architect.” [Online]. Available: http://www.sparxsystems.com.au/products/ea. [Accessed: 22-Mar-2017].

[14] Objecteering, “Objecteering, the model driven development tool.” [Online]. Available: http://www.objecteering.com/. [Accessed: 15-Feb-2017].

[15] Modeliosoft, the open source modeling environment, “Modelio.” [Online]. Available: https://www.modelio.org/. [Accessed: 25-Mar-2017].

[16] A. Lanusse *et al.*, “Papyrus UML: an open source toolset for MDA,” in *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, 2009, pp. 1–4.

[17] “BoUML.” [Online]. Available: http://www.bouml.fr/. [Accessed: 08-Mar-2017].

[18] Obeo, “UML Designer,” 2017. [Online]. Available: http://www.umldesigner.org/overview/. [Accessed: 25-Mar-2017].

[19] T. Mens and P. Van Gorp, “A taxonomy of model transformation,” *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, 2006.

[20] N. Koch, "Classification of model transformation techniques used in UML-based Web engineering," *IET Softw.*, vol. 1, no. 3, pp. 98–111, 2007.

[21] A. Kalnins, J. Barzdins, and E. Celms, "Model transformation language MOLA," in *Model Driven Architecture*, Springer, 2005, pp. 62–76.

[22] P. Tonella, "Reverse engineering of object oriented code," in *Proceedings of the 27th international conference on Software engineering*, St. Louis, MO, USA, 2005, pp. 724–725.

[23] L. Martinez, C. Pereira, and L. Favre, "Reverse Engineering Activity Diagrams from Object Oriented Code: An MDA-Based Approach," *Comput. Technol. Appl.*, vol. 2, no. 012, pp. 969–978, 2011.

[24] M. Usman and A. Nadeem, "Automatic generation of Java code from UML diagrams using UJECTOR," *Int. J. Softw. Eng. Its Appl.*, vol. 3, no. 2, pp. 21–37, 2009.

[25] M. Rahmouni and S. Mbarki, "Combining UML Class and Activity Diagrams for MDA Generation of MVC 2 Web Applications," *Int. Rev. Comput. Softw. IRECOS*, vol. 8, no. 4, 2013.

[26] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1–2, pp. 31–39, Jun. 2008.

[27] Object Management Group, "OMG Unified Modeling Language TM (OMG UML), superstructure." 2007.

[28] Eclipse Foundation, "Eclipse Java Developement Tools," 2017. [Online]. Available: http://www.eclipse.org/jdt/. [Accessed: 25-Mar-2017].

[29] "Java Metamodel." [Online]. Available: http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.modisco.java.doc%2Fmediawiki%2Fjava_metamodel%2Fuser.html. [Accessed: 18-Mar-2017].

[30] Object Management Group, "OMG Meta Object Facility (MOF) Core Specification." 2013.

[31] Object Management Group, "MOF 2 XMI Mapping, Version 2.4." 2010.

[32] S. J. Metsker and W. C. Wake, *Design patterns in java*. Addison-Wesley Professional, 2006.

[33] Object Management Group, "Semantics of a Foundational Subset for Executable UML Models (fUML)." 16-Jan-2016.

[34] Object Management Group, "Action Language for Foundational UML (Alf) Concrete Syntax for a UML Action Language Version 1.0.1." 09-Jan-2013.

[35] E. Seidewitz and J. Tatibouet, "Tool Paper: Combining Alf and UML in Modeling Tools–An Example with Papyrus–," in *OCL 2015–15th International Workshop on OCL and Textual Modeling: Tools and Textual Model Transformations Workshop Proceedings*, 2015, p. 105.

[36] A. Bergmayr, H. Bruneliere, J. Cabot, J. García, T. Mayerhofer, and M. Wimmer, "fREX: fUML-based reverse engineering of executable behavior for software dynamic analysis," 2016, pp. 20–26.

[37] B. Walters and V. Jones, "Middle school experience with visual programming environments," in *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*, 2015, pp. 133–137.

[38] A. Elmounadi, N. Berbiche, F. Guerouate, and N. Sefiani, "Smart Text to Model Transformation a Graph Based Approach to Cover Dynamic Analysis," *Int. Rev. Comput. Softw. IRECOS*, vol. 11, no. 4, p. 344, Apr. 2016.