The Journal of Supercomputing (2022) 78:7677–7699
https://doi.org/10.1007/s11227-021-04164-x

A novel approach with an extensive case study and experiment for automatic code generation from the XMI schema Of UML models

Anand Deva Durai[1] · Mythily Ganesh[2] · Rincy Merlin Mathew[3] · Dinesh Kumar Anguraj[4]

Accepted: 19 October 2021 / Published online: 3 January 2022

Abstract
Software models at different levels of abstraction and from different perspectives contribute to the creation of compilable code in the implementation phase of the SDLC. Traditionally, the development of the code is a human-intensive act and prone to misinterpretation and defects. The defect elimination process is again an arduous time-consuming task with increased time-to-deliver and cost. Hence, a novel approach is proposed to generate the code with the activity diagram and sequence diagram as the focus. The activity diagram and sequence diagrams and are defined as part of the UML definition to define the object flow of the system and interaction between the objects, respectively. An XMI schema is a text representation of any software model that is exported from a modeling tool. The modeling tool BoUML exports the required schema from the given input models such as sequence diagrams and activity diagrams. The proposed JC_Gen extracts artifacts from the XMI schema of these two models to generate the code automatically. The focus is mainly on class definition, member declaration, methods' definition, and function call in generated code.

Keywords XMI schema · Automatic code generation · UML diagram

✉ Mythily Ganesh
mythily.m@gmail.com

[1] College of Computer Science, King Khalid University, Abha, Saudi Arabia

[2] Department of Computer Science and Engineering, Karunya Institute of Technology and Sciences, Coimbatore, Tamilnadu, India

[3] Department of Computer Science and Information System, College of Science & Arts, King Khalid University, Abha, Saudi Arabia

[4] Department of Computer Science and Engineering, KL University, Vaddeswaram, Andhra Pradesh, India

1 Introduction

Automatic code generation is a typical need for the industry to avoid time delays on project delivery. The challenge is converting the model to text and extracting the proper properties of the model. UML is the de facto standard for modeling and design of software systems [1] in terms of structure and behavior. Thus, UML diagrams are largely classified into structural diagrams and behavioral diagrams. These diagrams are the high-level abstractions of the system.

The structural diagrams focus on the static structural aspects like entities and their relations to the system. The diagrams such as class diagrams, component diagrams, deployment diagrams fall into a structural category. The behavioral diagrams depict the dynamic nature of the system. The diagrams such as sequence diagrams, use-case diagrams, collaboration diagrams, state chart diagrams, and activity diagrams fall into the behavioral category. A sequence diagram is an interaction diagram that deals with the sequence of message exchanges between one object and another. An activity diagram is one of the significant behavioral modeling diagrams. It is the only UML diagram to represent the control flow (workflow) of the business process. JC_Gen uses a sequence diagram to generate the skeletal code and then the activity diagram to incorporate the business logic into the skeletal code generated earlier.

To obtain the perfect code, the association between the activity diagram and sequence diagram is needed, which represent object interactions and their behavior. Even though activity and sequence diagrams are behavioral models, they represent two different perspectives of the same system. This eases the code generation process of JC_Gen.

BoUml is the modeling tool that supports drawing large-scale models. It also runs on different platforms. To a greater extent, the XMI schema of the UML models is produced through this tool.

In this article, part 2 represents a detailed study on various aspects of the proposed system. Part 3 details the complete methodology and algorithms of the suggested system. Part 4 has given a complete case and step-by-step output/result production of the system. Finally, part 5 includes the future extension possibilities.

2 Review of related publications

The data collection of research has stepped into different dimensions such as, software modeling, model transformation, code generation, XMI tools. In this section, consolidation of the search is presented based on the dimensions. To note, code generation is a sub-category of model transformation.

UML is a standardized approach to represent models in the field of software engineering. It represents a set of graphic notations to create visual models of object-oriented software-intensive systems. There are a variety of applications that uses models as backbone are published in recent years. A model-based aspect-oriented framework [2] is proposed for building intrusion-aware software systems. In [3], proposes an aspect-oriented modeling (AOM) for incorporating security mechanisms in an application. Kong proposes [4], a graph grammar to summarize the hierarchy of states. The execution of a set of non-conflicting state transitions is predicted by a sequence of graph transformations. A group of experiments [5] investigates whether the use of stereotypes improves the comprehension of UML sequence diagrams. Even Babenko describes [6] a concept of information support as reusable. The proposed system also uses the UML Models to produce the code. A methodology [7] in which the aspect-oriented modeling AOM technique is used to customize the primary model by integrating different business requirements.

Model transformation of the MDA approach focuses on considering a model as an initiator and generating other models or the programming code automatically. Process model of SDLC decides the project management and determines the schedule, cost, time, and resource according to their liability. The model transformation approach eases the designing phase that saves effort and reduces errors by automating the building of other models as per the need. It also performs a better role in change management, where changes can be done in a single model and its impact will be injected into other models automatically. CIM (Computational Independent Business Model) to PIM (Platform Independent Behavioral Model) transformation is a mandate operation that converts the business view of a model into an information view [8]. This brings the need for information on business logic on model representations. There are a variety of algorithms [9] and prototypes [10]on model transformation to make an auto-conversion. Model transformation is classified as unidirectional, bidirectional, declarative, imperative, and rules [11]. There are many references to demonstrate how model transformation is achieved through different approaches. The article [12] suggests model transformation from the class model to relational model transformation with the help of model-driven engineering (MDE) principles. A generalized approach of mapping guidelines for CIM- high-level business model into PIM-low-level independent behavioral model is defined by [8]. A matching algorithm was proposed by [13] to convert a particular structure of source to destination model. Models can be transformed with different cardinalities like one to many models or many to one model, etc. [10]suggests a transformation approach of ATLAS model language to different modeling languages.

UML statechart diagram is used for modeling a system's dynamic behavior. [14] described an object-oriented approach for generating compact and efficient Java code from the statechart diagram. The states are represented as objects and all the behaviors associated with the states are retained as another set of objects. State design patterns have been extended with the help of object composition and delegation. JCode follows this approach to generate Java code after reading the specifications of the UML statechart diagram.

In their previous research [15], described a methodology where each state in the statechart has a class that encapsulates all the transitions and actions of the state. A readable, compact, and efficient code can be generated in the case of states without using controls such as if and case statements. Also, they published [16], produced executable legible, efficient, and compact code of the state diagram to an object-oriented language like Java. Representing states like objects that stretch out the hierarchical states' representation using the concept of composition of objects and delegation.

The gaps between the modeling and high-level programming languages are an obstacle to produce satisfactory solutions. The automation tool proposed by [17] addresses this issue by mapping the UML notations to Java. It can generate directly the high-level Java code from multiple UML statecharts. It suggested a process of requirements engineering that composes UML scenarios to obtain a comprehensive description of a given service system. The derived services are transformed into the source code. Four operators are suggested as, sequential, competing, conditional, and iteration operators to compose a set of scenarios that describe the use case of a given system.

An [18] investigation on the viability of automatic generation of code from current systems design is taken place. Several different approaches have been experimented with in terms of short-duration and futuristic approaches.

Usman and Nadeem [19] extended their work on a tool called UJECTOR [19] for the automatic generation of executable Java code from UML diagrams. A set of three UML diagrams, i.e., class diagram, sequence diagram, and activity diagram are the input to the tool to generate a completely executable Java code automatically.

Parada et al. [20] presented a work to automatically generate structural and behavioral code from UML class and sequence diagrams. In [21], Engels et al. concentrated on collaboration diagrams. The main objective is automatically generated java code fragments to build a substantial part of the system's functionality and to avoid the loss of important information during the transformation process.

A comparison study on generated code [22] from rhapsody OPCAT, using object-process methodology (OPM) case tool. The comparison concludes that the UML consistency problem and its distributed representation of the system behavior are reflected in the code. OPM models, which capture the static and dynamic aspects of a system in a single view, also enable the generation of potentially complete application logic rather than just skeleton code. The study also explained the unique architecture and functionality of OPM- GCG (Generic Code Generator) of OPCAT.

The research mainly focused on bridging the gap between software design and implementation. Same as indented by [23], the systems-based components are used in software architecture at the level of modeling/design. Then, the coordination paradigm components are used at the level of implementation.

In Singh has proposed [24], UML class diagram is used to generate XML (Extended Markup Language) schema. The generated XML schema is used for code generation. JIBX (Binding XML to Java code) is a Java-based open-source tool used for code generation which is developed by IBM.

Gene-Auto ITEA European project [25], which aims at building a qualified C code generator from mathematical models under MATLAB-Simulink and Scilab-Scicos. The first version of the Gene-Auto code generator has already been released and has gone for a validation phase on real-life case studies defined by each project partner.

Automation of model generation improves the reusability of the software development process. This process is intended in designing the pictorial model of the information and exporting it as a schema. Some of these tools assist the conversion of the desired format of input and output of the system. ArgoUML is an object-oriented case tool that facilitates the generation of a source model. ArgoUML contains multiple functionalities for the UML model generation, and it exports the XMI of the model [26].

A new approach of auto-code generation is proposed in [27], which uses the Rete algorithm to generate rapid code generation using a uniform coding style.

BoUml is a modeling tool to support large-scale models and is easy to draw the model. It also runs under different platforms. To a great extent, the XML schema of the UML models produces through the tool [28]. The exported XML schemas are taken as the source to develop the applications in Java [29]

The decisions about the functionality and structure of any software system are so critical and decided at the design phase and the design of XML schemas involved in the development as a consequence of those decisions. In [30] proposed, transformation algorithm to convert a UML profile into XML schema. This model-driven approach allows designers to be freed from low-level implementation issues, by the fully automatic mechanisms that transform UML models to XML schemas.

A learning system analysis metrics value from the existing system ad uses principal feature analysis to find the complexity, cohesion, and coupling is addressed [31].

A tool called PlantUML is a flow charter or UML model generator that takes rendered text as input to generate specific models according to the given text annotations [31]. Rendering is a process of joining the entire element and their relationship to form the standard textual format so that the tool can regenerate images from text [32].

Graphviz is an open-source tool that is used for generating graphical representations described in [33]. For the graphical representation, the design patterns and dot tool of the Graphviz package are used in [34]. Object Relation Diagram (ORD) is a directed graph, where nodes are classes and the edges represent the relationship between classes, and advantage of this approach is that it reduces the cost of stub creation.

3 Java-specific code generator (JC_Gen)

The proposed tool JC_Gen makes use of sequence diagrams and activity diagrams in generating code. Parsers segments data from the XMI into categories like action, activity, model, state, transition, etc. Some of the Java APIs like DOM, SAX, dom4j, and XOM are used for checking and validating the XML with DTD and Schema(s). A DTD is a document type definition that defines the structure, legal elements, and attributes of an XML document.

Kraft et al. [35] perform a comparative study on various parsers concerning their parsing time and throughput. This involves a set of stream-based parser APIs such as SAX (Simple API for XML), StAX, XMLPull, and tree-based APIs such as document object model (DOM), JDOM, ElectricXML, DOM4j. Another deliberate discussion on Java APIs of XML parser is done in [36].

A sequence diagram shows object interactions arranged in the time sequence of a system. It depicts the objects involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.

An activity in Unified Modeling Language (UML) is a major task of object-oriented development that defines an activity as a sequence of activities that make up a process.

The architecture of JC_Gen is shown in Fig. 1 which outlines the flow of processes performed in obtaining the java code. Using the above models, the major perspective of the coding can be retrieved. Especially sequence helps to identify classes, data members, calling function, and called function. The activity model fills the logic of the declared function. Therefore, these model suits well to generate code automatically.

The following steps are involved in code generation.

1. XMI generation
2. SD Parsing (Sequence Diagram Parsing)
3. AD Parsing (Activity Diagram Parsing)
4. Structural code generation
5. Behavioral code generation

    a. Mapping between sequence and class artifacts

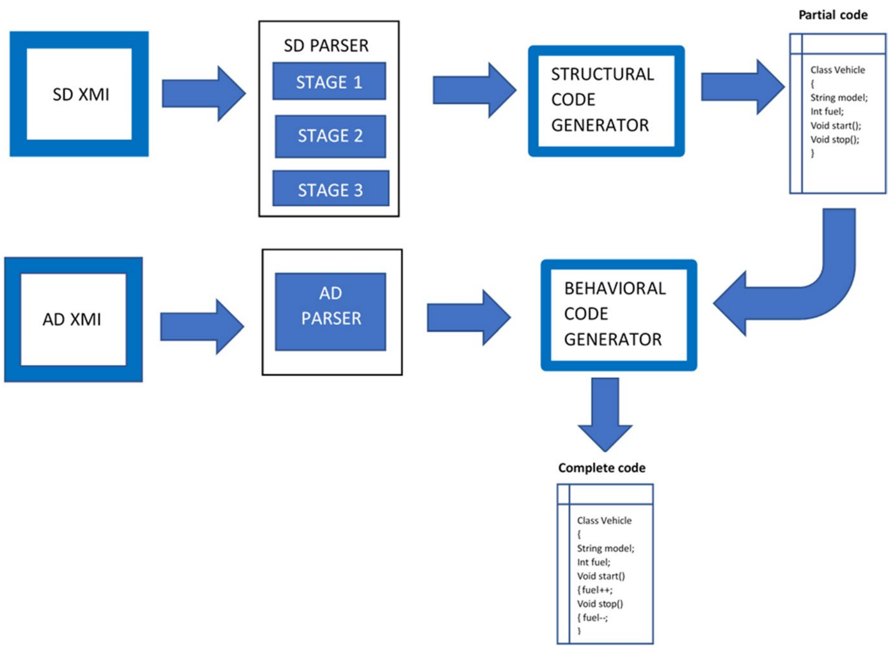

Fig. 1 Architecture of JC_Gen

b. Activity Interpretation

The XMI schemas of both sequence diagram and activity diagram form the input to the respective parsers. The SD parser retrieves the information of structural aspects such as class definition and members' declaration from the SD XMI schema. The output of the SD parser is used to produce the partial code of the software by the structural code generator. The AD parser retrieves the logic of each member in the form of activity in terms of action and flows. The behavior code generator takes structural code and extracted elements from activity diagram to incorporate the business logic code in the generated code.

3.1 XMI generation

The XMI schema generated by the modeling tool represents the behavior of the model elements in the form of tags and their attributes. These elements are also organized according to the order of their structure and action. The extraction of the sequence diagram is intended to gather the structural information of the code. However, the Activity XMI representation focuses in-depth on each activity and its sub-actions to derive the behavior aspects of the code. Dependency issues may raise with XMI version usage due to the update of schema versioning. Until now the experiment was carried by carefully using compatible schemas. Also, an alert message is provided when the version is not supported by the developed system.

In the sequence diagram, each lifeline represents the object's interaction time ordered. The message passed between the objects decides the flow of the program. Messages are the communication between objects. Each message reflects either an invocation of a method or sending and receiving of a signal, so that message could be of actions like synchronous call, asynchronous call, and asynchronous signal [37]. Each message specification has sender and receiver in which the sender will be calling or passing a message which resides on the receiver side. In a sequence diagram, each message can be considered as part of the sequence flow, from one object to another object over the timeline.

The activity diagram identifies the procedure of each method by its action and sub-actions. The start and the top nodes of the activity delimit the scope of the method specified. The action state provides a pseudovalue of the procedure rather than a java code. These actions are mapped with the sequence diagram messages to produce method definitions of corresponding members.

3.2 SD parser

The SD parser extracts the necessary elements from the XMI document. The extracted elements are segregated according to their type value such as class, property, operations, and messages. The relationships between the elements are maintained to generate the code. To achieve the extraction of elements and their relationship in an optimized way, the parenthesis balancing algorithm is used [38]

XMI tags, the foundation of XMI define the scope of an element in XMI. They can also be used to insert comments, declare settings required for parsing the environment, and insert special instructions. XMI tags are represented in the form of tree structure. Each tag has some relevant sub-tags. It has the facility to store the property of each tag. For example,

```
<book category="education">
<title lang="en">Complete                                    Reference</title>
<author>Herbert                                              Schilt</author>
<year>2020</year>
<price>899</price>
</book>
```

It is found storage of extracted elements with their members and relationships can be obtained efficiently by using tree structure storage. Therefore, the output of the SD stage 1 parser is a tree with a set of nodes representing the XMI tags. Each node retains its name, type, and xmi:id. It parses only the open and close type tags. The algorithm used to create the class structure with its data members, and member function is given in Algorithm 1.

Algorithm 1: SD stage 1 Parser

```
Input: XMI Schema
Output: Tree structure T
//XMI document is given as input and a tree is formed with the help of Stack.
Stack S –linear storage of the type node.
T_node – an object of a node with a name and type field.
T, open_T, close_T, and Combined_T- tags extracted at a particular iteration.
1. Repeat steps 2 and 3 until no more Tg
2. If Tg is open_T$_i$ then create a node T_node
3.         Extract the T_type$_i$, T_name$_i$, and XMI: id field to initialize the node.
4.         If T_root is null then T_node$_i$ becomes T_root.
5.         S_Push() the T_node$_i$ into the stack S.
6.         Insert T_node$_i$ as a child of the S_top node.
7. If the is close_T$_i$ then
8.         Perform S_pop()
9. End
```

The SD parser stage 1 generates a tree structure using a parenthesis balancing algorithm. This is adopted and modified in the work to identify each pair of tags

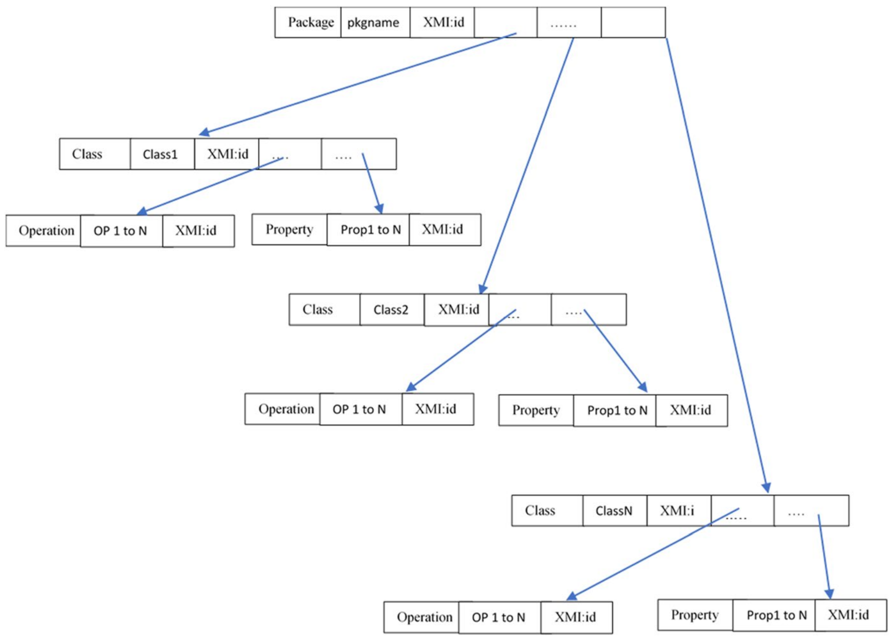

Fig. 2 Tree structure generated by SD stage 1 parser

presented in the XMI Schema. Through this, the tree structure is formed by the Stage 1 SD parser.

It determines the hierarchies of the classes and the data members as shown in Fig. 2. This algorithm neglects the combined tag which only focuses on the collaboration and more details of the model than the core components. The tree structure thus produced by the SD stage 1 parser along with XMI schema is used to produce mtd_Array and var_Array by the SD stage 2 parser.

The datatype and its initial value of the data members and prototype of each method with its return type are obtained by the SD stage 2 parser using algorithm 2. The SD stage 1 tree along with the XMI document is given as input to this algorithm. The "operation" and "property" nodes of tree 'T' are processed repeatedly to find this type. The output of the algorithm is stored in a sequence structure for further processing. The SD stage 2 parser produces two structures to store data members (var_Array) and member functions with (met_def) data types and return types, respectively.

Algorithm 2 SD stage 2 Parser

```
Input: XMI Schema and Tree Structure T
Output: mtd_Array and var_Array
    1. Repeat steps 2 and 3 for all member nodes of the class.
    2. For all the identified “operation” type node $T\_node_i$ (Member function)
        a. Identify the <ownedOperation> tag where name=$T\_node_i$ :name
        b. Extract the inner <Owned parameter> tag of <owned operation> and
           check the name is return then go to step 3.
        c. Extract the inner <type> tag of corresponding <ownedOperation> and
           identify the xmi:idref value.
        d. Identify the <packagedElement> tag where its xmi:id is equal to
           xmi:idref of step ‘c’.
        e. Return the name attribute of <packagedElement> (i.e., the return type
           of $T\_node_i$)
        f. Store in mtd_arr as the method name and return type.
    3. For all the identified “property” type node $T\_node_i$ (Data Members)
        a. Identify <ownedAttribute> tag where name=$T\_node_i$ :name
        b. Extract the inner <defaultValue> tag of corresponding
           <ownedAttribute> and extract the initial value of member $T\_node_i$
           from value attribute
        c. Extract the inner <type1> tag of corresponding <ownedAttribute> and
           identify the xmi:idref value.
        d. Identify the <packagedElement> tag where its xmi:id is xmi:idref of
           step ‘c’.
        e. Return the name attribute of <packagedElement> (i.e., the data type of
           $T\_node_i$)
        f. Store in var_Array as variable_name and data_type.
```

The SD stage 3 parser extracts the sequence of object flow among the classes using algorithm 3. The algorithm SD stage 3 uses the same inputs used for the stage 2 parser. The output of the SD stage 3 parser is used to frame the function call inside the main function of the classes according to the source sequence model.

Algorithm 3 SD stage 3 Parser

```
Input: XMI Schema and Tree Structure T
Output: objFlow_array
  1. Repeat step 2 for all the identified class type node $T\_node_i$
  2. Repeat steps 3 to 7 until all method calls are retrieved
  3. Identify <ownedAttribute> tag where its "type" attribute is $xmi{:}id_{class}$
  4. Identify <lifeline> tag where its "represents" attribute is $xmi{:}id_{ownedAttribute}$
  5. Identify <fragment xmi:type="uml:MessageOccurrenceSpecification"> tag
     where its "covered" attribute is $xmi_{lifeline.}$
  6. Identify <message> tag where its sendEvent attribute is xmi:id
     $_{MessageOccurrenceSpecification}$
  7. Return the name of the operation and the caller class which requested the
     operation. The receive event of the message identifies the method owner by
     performing the step 1 to 5 in reverse order. The message tag provides the name
     of the operation to be called in the $xmi{:}id_{class}$.
  8. Store in obj_Array with structure element Ele (function name, owner class,
     and caller class) which represent the object flow between classes.
```

The 3 stages of the SD parser generate a set of data structures such as T, mtd_Array, var_Array, and objFlow_array. These details produce an array of structural information to generate the code.

3.3 AD parser

AD parser helps to extract the behavior aspects of the software under development. It considers the XMI schema of the activity model generated corresponding to the sequence model. Multiple discontinuous activity models are used to represent the tasks performed inside the methods.

Each activity delimits the other activities by start and stop nodes. In between nodes depict the logic of the operation. The output of the AD parser is stored in an array where each element is the head of a singly linked list. Each singly linked list is used to represent the execution sequences of an operation. AD parser incorporates the methods defined in the appropriate place of the generated code by the SD parser. Algorithm 4 represents the AD parser. The element in mtd_def_Array shown in Fig. 6 represents the logic of each method in its sequence.

Algorithm 4 AD Parser

```
Input: XMI Representation of Activity Diagram
Output: mtd_def_Array
    1. Repeat steps 1 to 4 for each <packagedElement>$_i$ in XMI.
    2. If XMI: type is Activity create a head node and store the name in the head
       node.
    3. Store head into an i$^{th}$ position of the mtd_def_Array.
    4. Repeat steps 5 to 8 for each <edge>$_j$ in <packagedElement>.
    5. If XMI: type is control-flow then find the id of source and target nodes.
    6. If the source and target nodes are found in i$^{th}$ position of the array's list then
       create a node with "loopend" as data and insert it at the end of the list.
    7. If the source node is not found in i$^{th}$ position of the array's list then create a
       new node with the source's name and insert it at the end of the list.
    8. If a target node is not found in i$^{th}$ position of the array's list then create a new
       node with the target's name and insert it at the end of the list.
```

3.3.1 Structural code generator

The output of three SD stage parsers is given as an input to the structural code generator. SD stage 1 parser defines the classes and their data members. The tree structure 'T' indicates the relation between the different types of elements. SD Stage 2 parser identifies the variable and returns types of members, and finally, the SD stage 3 parser represents the function calls. This process is a clear mapping of parser extracted elements into the code statements. Each element of tree structure 'T' is fixed into the code structure according to its position and type. (e.g., Each class type element is created as a Class and the child of these elements in 'T' are its members. The data members and member function are identified based on the type.) Then, mtd_Array produces the return type and var_Array produces data type and initial value.

objFlow_array indicates the class where the function is to be called and the class to which the function belongs. Using this data, the function calling statements are transformed to code structure. The template code generator [39] adheres to the information by the parser to produce the java code. The structural code generator produces the class definition inclusive of variable and method declaration. It also focuses on the function calling to represent the object flow from one class to another class. When the array is combined with the result of the structural code generator will produce the compilable code of the given input models.

3.3.2 Behavioral code generator

This process updates the already generated code with method definition. The method definition is a procedure of task accomplishment in the software. This behavior is extracted from the activity diagram by the AD parser and stored in mtd_def_Array in Fig. 6. Each element stores a particular method's definition in the form of a linked list. During this incorporation of definitions, the following sub-tasks are performed.

3.3.2.1 Mapping between sequence and class artifacts This mapping identifies the appropriate place to incorporate the method definition. If it matches the function name of the code generated with the element name of mtd_def_Array in Fig. 6. If there is a matching found, then the entire linked list with the same element name is inserted inside the block below the matched node as in the sequence Tree '*T*'. Set of pseudo-codes are produced as the method definitions, and it will be modified by the activity interpreter.

Activity interpreter. The pseudo-code is replaced by java code by activity interpreter. The pseudo-statements of method definition are converted to java code with the help of an interpreter. The start and final statements are replaced with '{'and '}' by the activity interpreter. Hence, the scope of the method is determined. The pseudo-code is treated as a keyword, and the replacement is done for that statement. E.g., "add a, b" in the pseudo-code is replaced by "a = a + b;" otherwise "calculate c = a + b;" replaced by "c = a + b;". The output of this process produces the pure java code.

Assumptions Some of the assumptions made after complete code generation are listed below.

- If any method declaration does not found a match in mtd_def_Array then treat it as an abstract function.
- If a class has one or more abstract functions, then the class is also considered abstract.
- If no method definitions are found in a class, then it is an interface.
- Method calling happens only in the main method until it is explicitly specified in the method definition.

4 Results and discussion

A case study on car driving has been taken as a sequence model with a set of activity diagrams. The car, driver, and engine are identified as the call actions and flow between these actions is captured by the corresponding objects. The input models are generated using the BOUML tool and an XMI schema of the corresponding source models is exported.

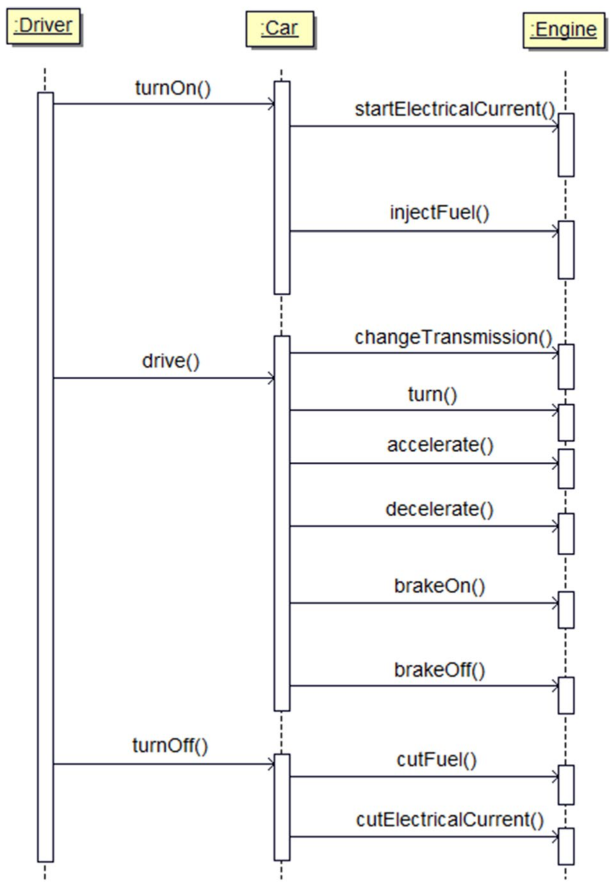

Fig. 3 Driving sequence diagram as source model

For the given sequence diagram in Fig. 3, the type of objects required (i.e., the corresponding classes) and their members are retrieved. A partial tree of the source model is shown in Fig. 4 which represents the call action "car" of the sequence model. The XMI schema exported from the modeling tool is further classified by different levels of parsers. This schema contains different tags with multiple attributes to reproduce the modeling diagram. The SD parser stage 1 generates a tree structure using a parenthesis balancing algorithm. This Tree structure is represented as 'T' in further processes and each node of this tree keeps track of its type, id, and link to the child. The elements are extracted from the XMI schema, and a tree is formed. The types of nodes such as class, operation, and property are retrieved by the SD stage 1 parser. It also determines the hierarchy of the elements as shown in Table 1. The members are incomplete without their return type and data types. The driving model consists of 3 classes such as driver, car, and engine.

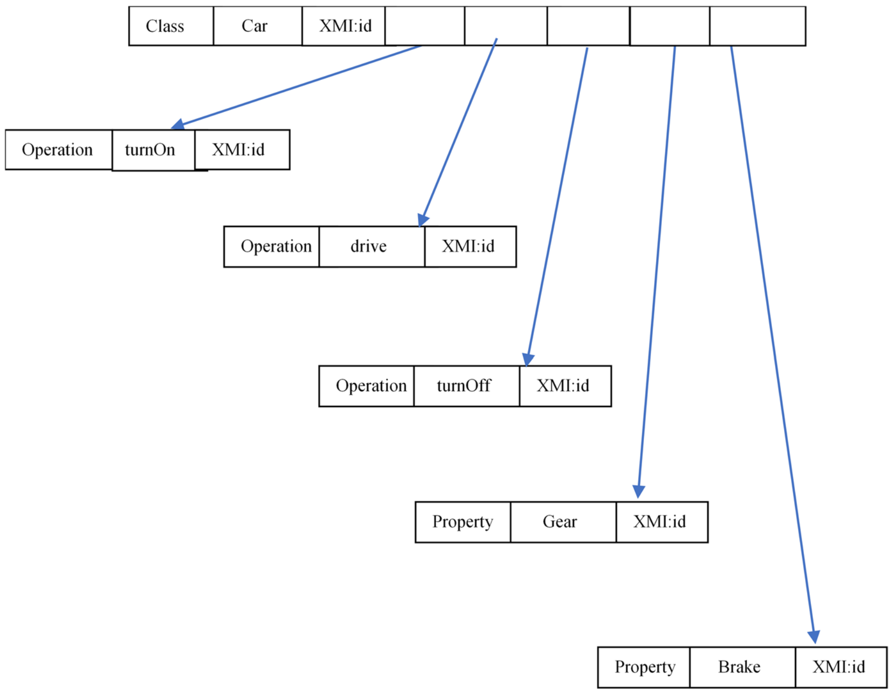

Fig. 4 'T' Structure of Car class and members

Table 1 Mtd_Array of driving model

Name	Return type
Turn on	Boolean
Drive	void
Turn off	Boolean
Start electrical current	Boolean
Inject fuel	void
Change transmission	void
Accelerate	void
Decelerate	void
Brake on	void
Brake off	void
Cut fuel	void
Cut electrical current	Boolean
Turn	void

To retrieve the missing information SD stage 2 parser takes the 'T' as input and traverses the entire array. If the node type is "operation," then the return type of function is identified and stored in mtd_Array; otherwise, if the node type is

Table 2 Var_Array of driving model

Name	Data type	Initial value
Gear	int	False
Brake	Int	0
Ignition	Boolean	False
Fuel	int	0
Gear	Int	0
Brake	Int	0
Current	Int	1
Turn	Boolean	False

"property," then data type and the initial value of the member identified are stored in var_Array. Two arrays are produced by the SD stage 2 parser concerning the source sequence model and are shown in Tables 1 and 2.

The next level of the parser is used to find the object flow from one class to another class with the help of message flow. This SD stage 3 parser determines the place of the function call and creates the object used to call the function. The different function calls of each class are identified and inserted into the main function. According to the identified owner class, the objects are created to utilize the functions. Even though there is more than one class in a source code file, only one main function can exist in it. Hence, the main function is created under the initial class with message flow in the model. The output of the SD stage 3 parser is stored in objFlow_Array as shown in Table 3.

After the SD stage parsers of structural code are generated by the structural code generator, it converts the data stored in mtd_Array, var_Array, and objFlow_array into the code format. Each class structure is stored in a separate file with the class

Table 3 Objflow_Array in driving model

Name	Owner class	Caller class
Turn on	Car	Driver
Drive	Car	Driver
Turn off	Car	Driver
Start electrical current	Engine	Car
Inject fuel	Engine	Car
Change transmission	Engine	Car
Accelerate	Engine	Car
Decelerate	Engine	Car
Brake on	Engine	Car
Brake off	Engine	Car
Cut fuel	Engine	Car
Cut electrical current	Engine	Car
Turn	Engine	Car

Table 4 Code obtainment from SD stage parsers

Code after SD Stage1 Parser	Code after SD Stage 2 Parser	Code after SD Stage 3 Parser
public class Driver	public class Driver	public class Driver{
{	{	public static void main(String args[])
}	}	{
public class Car	public class Car{	Driver obj1 = new Driver();
{	public int Gear = 0;	Car car_obj = new Car();
Gear;	public int Brake = 0;	car_obj.turnOn();
Brake;	public boolean turnOn()	car_obj.drive();
turnOn();	{	car_obj.turnOff();
drive();	}	Engine Engine_obj = new Engine();
turnOff();	public boolean drive()	Engine_obj.startElectricalCurrent();
}	{ }	Engine_obj.injectFuel();
public class Engine{	public void turnOff()	Engine_obj.changeTransmission();
Ignition;	{ }	Engine_obj.turn();
Fuel;	}	Engine_obj.accelerate();
Gear	public class Engine	Engine_obj.decelerate();
Brake;	{	Engine_obj.brakeOn();
Current;	public boolean Ignition = False;	Engine_obj.brakeOff();
Turn;	public int Fuel = 0;	Engine_obj.cutFuel();
ElectricalCurrent();	public int Gear = 0;	Engine_obj.cutElectricalCurrent();} }
injectFuel();	private int Brake = 0;	public class Car{
changeTransmission();	public int Current = 1;	public int Gear = 0;
accelerate();	public boolean Turn = False;	public int Brake = 0;
decelerate();	public boolean startElectricalCur-	public Boolean turnOn(){ }
brakeOn();	rent()	public void drive(){ }
brakeOff();	{ }	public Boolean turnOff(){ } }
cutFuel();	public void injectFuel()	public class Engine {
cutElectricalCurrent();	{ }	public boolean Ignition = False;
turn();	public void changeTransmission()	public int Fuel = 0;
}	{ }	public int Gear = 0;
	public void accelerate()	private int Brake = 0;
	{ }	public int Current = 1;
	public void decelerate()	public boolean Turn = False;
	{ }	public boolean startElectricalCur-
	public void brakeOn()	rent(){ }
	{ }	public void injectFuel(){ }
	public void brakeOff()	public void changeTransmission(){ }
	{ }	public void accelerate(){ }
	public void cutFuel()	public void decelerate(){ }
	{ }	public void brakeOn(){ }
	public boolean cutElectricalCur-	public void brakeOff(){ }
	rent()	public void cutFuel(){ }
	{ }	public boolean cutElectricalCur-
	public void turn()	rent(){ }
	{ }	public void turn()}
	}	

name. The stage-by-stage code achievement concerning SD parsers is shown in Table 4.

The behavior aspects of each operation are represented by another input model called activity diagram as shown in Fig. 5. The input activity diagram represents procedures for each method with a set of states. The set of activities of a method

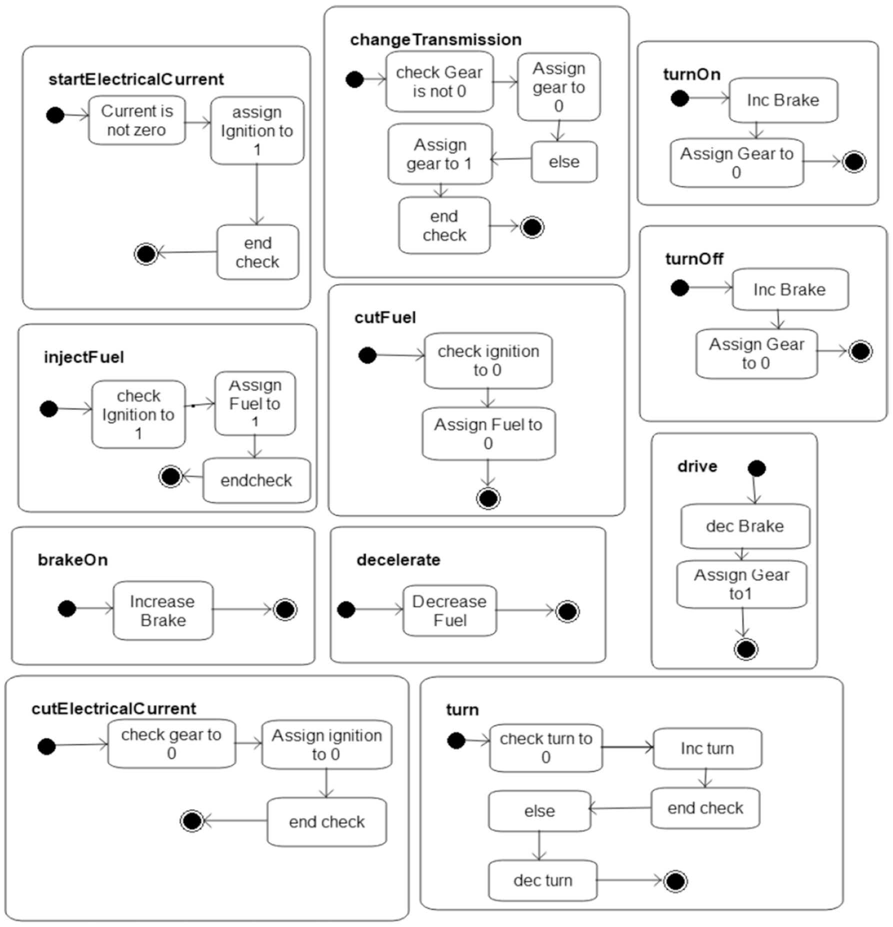

Fig. 5 Driving activity model as source model

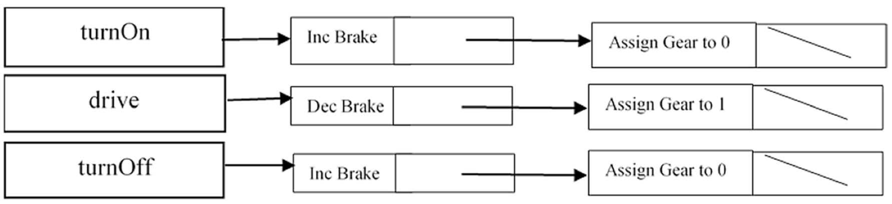

Fig. 6 Mtd_def_Array with an activity list of Car class

is grouped with a distinct name. The AD parser extracts the elements from the XMI schema and stores each activity in a separate list. The head of each list is stored in an array.

The array with car class methods is shown in Fig. 6. The partition of the activity model is mapped with the mtd_Array of SD stage 2 parser. If any matching occurs, then the activity list is inserted as the definition of the member function.

After the mapping, the behavior code remains as pseudo-code. To address this issue, an activity interpreter is used to convert the pseudo-code into java code. This activity interpreter makes the code a platform-specific model such as Java code. A sample is shown in Table 5. This reduces the burden of the designer on code generation.

Finally, a complete java code that represents both structural and behavioral aspects of the models is produced. Through the source models, three classes are achieved in the given case study as driver, car, and engine. The complete Java code in Table 6 shows the compilable code placed in the given input models. Each class is stored in a separate file, and the output of the main class is shown in Table 6. Similar attempt of generating code was attempted earlier using different diagrams [40].

LPMT [13] is a logical prediction model transformation system that takes the activity diagram as the source then produces a class diagram, use-case diagram, and sequence diagrams. Extracts information from an XMI schema and applies rules on the source model to destination models.

A comparison is between JC_Gen and LPMT model proposed in [LPMT] is shown in Table 7. The table depicts the correlation among them since they are different forms of model transformation. The software metrics [41] do not reveal the accomplishment of code generation from the models. But code generation approach has proven the time reduction of code generation and only a few more contribution expected from the human.

5 Conclusion

This initiative is a stepping stone in the development of java applications from the models. The procedures followed here can also be extended to develop GUI programs using java without the knowledge of syntax. The final code produced by the work is 95% compliable code. This research reduces the gap between design models and code generation. It strengthens the connectivity between the phases of the SDLC process. It supports large-scale models compared to other flowchart interpreters such as raptor, flowgorithm, visirule, etc., and concentrate on business aspects

Table 5 Sample interpreter table

Pseudocode	Sample	Java code
Initialize < var > to < val > / Assign < var > to < val >	Assign Gear to 0	var = val
Check < var > to/not/grt/less < val >	Check fuel less 10	If(var = /!/ > / < val)
Inc < var >	Inc Brake	var + +;
pInc < var >	pInc Accelerator	+ + var;
Calc < expr >	Calc B = 3.14*rad*rad	expr;

Table 6 The output of the AD parser

Driver	Car	Engine
public class Driver	Public Class Car	Public class Engine
{	{	{public boolean Ignition=false;
public static void main(String args[])	public int Gear=0;	public int Fuel=0;
{	public int Brake=0;	public int Gear=0;
Driver obj1=new Driver();	public int turnOn()	private int Brake=0;
Car car_obj=new Car();	{	public int Current=1;
car_obj.turnOn();	Brake++;	public int Turn=0;
car_obj.drive();	Gear=0;	public boolean startElectricalCurrent(){
car_obj.turnOff();	System.out.println("turnOn");	if(Current!=0)
Engine Engine_obj=new Engine();	return Brake;	{Ignition=true;} System.out.println
Engine_obj.startElectricalCurrent();	}	("StartElectricalCurrent");
Engine_obj.injectFuel();	public void drive()	return Ignition;}
Engine_obj.changeTransmission();	{	public void injectFuel(){
Engine_obj.turn();	Brake–;	if(Ignition==true){
Engine_obj.accelerate();	Gear=1;	Fuel=1;}
Engine_obj.decelerate();	System.out.println("drive");	System.out.println("InjectFuel");}
Engine_obj.brakeOn();	}	public void changeTransmission()
Engine_obj.brakeOff();	public int turnOff()	{if(Gear!=0)
Engine_obj.cutFuel();	{	{Gear=0;}
Engine_obj.cutElectricalCurrent();	Brake++;	else{
}	System.out.println("turnoff");	Gear=1;} System.out.println("ChangeTra
}	return Brake;	nsmission");}
	}	public void accelerate()
	}	{Fuel++; System.out.
		println("accelerate");}
		public void decelerate()
		{Fuel–;
		System.out.println("deacelerate");}
		public void brakeOn()
		{Brake++;
		System.out.println("brakeon");}
		public void brakeOff(){
		Brake–;
		System.out.println("brakeoff");
		}
		public void cutFuel()
		{if(Ignition==false)
		{Fuel=0;}
		System.out.println("cutfuel");}
		public boolean cutElectricalCurrent(){
		if(Gear==0){
		Ignition=false;} System.out.println("cutEl
		ectricalCurrent");
		return Ignition;}
		public void turn()
		{if(Turn!=0){
		Turn++;}
		else{
		Turn–;}
		Gear=0;
		System.out.println("turn");}}1

more than case tools such as StarUML, BoUML, ArgoUML, etc. The addressing of behavioral aspects of the software using advanced models such as, activity and sequence diagram is an added advantage of this system as it is capable of producing

Table 7 Comparison between JC_Gen and LPMT

Parameter	LPMT	JC_Gen
XMI Schema	ArgoUML	BoUML
Parsers	XOM parser	SD and AD parser
Input	Sequence Model	Sequence and Activity
Output	Class and Activity model	Java Code
Methodology	Logical Predictor in Model Transformation	Template-based code generation
Transformation	PIM-M1to PIM-(M2 &M3)	PIM to PSM

the code with business logic when compiled produce necessary executables, which are production-ready.

However, this system can be extended to other UML models and the high-level concepts such as dynamic method dispatch, reusable design patterns, and inclusion of library function.

Acknowledgements This research is done with the financial support by the Deanship of Scientific Research at King Khalid University under research grant number RGP.1/210/42.

References

1. James B, "System Development Life Cycle (SDLC)—Risk Management Frammework." https://www.oreilly.com/library/view/risk-management-framework/9781597499958/B9781597499958000053.xhtml Accessed June 03 2021
2. Zhu ZJ, Zulkernine M (2009) A model-based aspect-oriented framework for building intrusion-aware software systems. Inf Softw Technol 51(5):865–875. https://doi.org/10.1016/j.infsof.2008.05.007
3. George MLV, Vadakkumcheril T, Mythily M (2013) A simple implementation of UML sequence diagram to java code generation through XMI representation. Int J Comput Theory Eng 3(12):35–41. https://doi.org/10.7763/IJCTE.2009.V1.6
4. Kong J, Zhang K, Dong J, Xu D (2009) Specifying behavioral semantics of UML diagrams through graph transformations. J Syst Softw 82(2):292–306. https://doi.org/10.1016/j.jss.2008.06.030
5. Cruz-Lemus JA, Genero M, Caivano D, Abrahão S, Insfrán E, Carsí JA (2010) Assessing the influence of stereotypes on the comprehension of UML sequence diagrams: a family of experiments. Inf Softw Technol 53(12):1391–1403. https://doi.org/10.1016/j.infsof.2011.07.002
6. Babenko LP (2003) UML-based software engineering. Cybernet Syst Anal 39(1):65–70
7. Zou Y, Xiao H, Chan B (2007) Weaving business requirements into model transformations. Business, 1–10
8. de Castro V, Marcos E, Vara JM (2011) Applying CIM-to-PIM model transformations for the service-oriented development of information systems. Inf Softw Technol 53(1):87–105. https://doi.org/10.1016/j.infsof.2010.09.002
9. Asztalos M, Lengyel L (2008) A metamodel-based matching algorithm for model transformations. Computational Cybernetics, 2008. ICCC 2008. IEEE International Conference on, pp 151–155
10. Sanchez Cuadrado J, Guerra E, de Lara J (2014) A component model for model transformations. IEEE Trans Softw Eng 40(11):1042–1060. https://doi.org/10.1109/TSE.2014.2339852
11. Czarnecki K, Helsen S (2003) Classification of model transformation approaches. pp 1–17
12. Bollati VA, Vara JM, Jiménez Á, Marcos E (2013) Applying MDE to the (semi-)automatic development of model transformations. Inf Softw Technol 55(4):699–718. https://doi.org/10.1016/j.infsof.2012.11.004

13. Mythily M, Valarmathi ML, Durai CAD (2018) Model transformation using logical prediction from sequence diagram: an experimental approach. Clust Comput. https://doi.org/10.1007/s10586-017-1618-5
14. Niaz IA, Tanaka J (2005) An object-oriented approach to generate java code from UML statecharts. Int J Comput Inf Sci 6(2):83–98
15. Niaz IA, Tanaka J, Mapping uml statecharts to java code
16. Niaz IA, Tanaka J Code generation from uml statecharts
17. Jakimi A, Elkoutbi M (2009) Automatic code generation from UML statechart. Int J Eng Technol 1(2):165–168
18. Burke PW, Sweany P (2007) Automatic code generation through model-driven design
19. Usman M, Nadeem A (2009) Automatic generation of java code from UML diagrams using UJECTOR. Int J Softw Eng Appl 3(2):21–37
20. Parada AG, Siegert E, de Brisolara LB (2011) Generating java code from UML class and sequence diagrams. Int J Comput Inf Sci, 99–101
21. Engels GW, Hücking R, Sauer S (1999) UML collaboration diagrams and their transformation to Java. In: International Conference on the Unified Modeling Language, pp 473–488. https://doi.org/10.1007/3-540-46852-8_34
22. Reinhartz-berger I, Dori D (2004) "Object-process methodology (OPM) vs. UML : a code generation perspective
23. Stavrou A, Papadopoulos GA (2007) Automatic generation of executable code from software architecture models. In: information system development, Springer, Boston, MA, 2007, pp 1–12. https://doi.org/10.1007/978-0-387-78578-3_36
24. Singh S (2012) Effort reduction by automatic code generation. Int J Comput Sci Eng Technol (IJCSET) 3(8):366–369
25. Rugina A, Thomas D, Olive X, Veran G (2008) Gene-auto : automatic software code generation for real-time embedded systems. In: proceedings of DASIA 2008 data systems in aerospace, no 1
26. Robbins JE, Redmiles DF (2000) Cognitive support, UML adherence, and XMI interchange in Argo/UML. Inf Softw Technol 42(2):79–89. https://doi.org/10.1016/S0950-5849(99)00083-X
27. Chen H (2020) Design and implementation of automatic code generation method based on model driven. J Phys Conf Series. https://doi.org/10.1088/1742-6596/1634/1/012019
28. Bruno, "BOUMLtutorial," 2011. http://www.bouml.fr/tutorial/tutorial.html
29. Barclay K, Savage AJ (2004) Object-oriented design with UML and java. Elsevier Butterworth-Heinemann, [Online]. Available: http://digilib.mercubuana.ac.id/manager/n!@file_ebook/Isi1964329425705.pdf
30. Domínguez E, Lloret J, Pérez B, Rodríguez Á, Rubio ÁL, Zapata MA (2011) Evolution of XML schemas and documents from stereotyped UML class models: a traceable approach. Inf Softw Technol 53(1):34–50. https://doi.org/10.1016/j.infsof.2010.08.001
31. Dimaridou V, Kyprianidis AC, Papamichail M, Diamantopoulos T, Symeonidis A (2019) Towards modeling the user-perceived quality of source code using static analysis metrics. ICSOFT 2017—Proceedings of the 12th International Conference On Software Technologies, no. March 2019, pp 73–84, 2017, https://doi.org/10.5220/0006420000730084
32. Kosower DA, Lopez-Villarejo JJ (2015) Flowgen: flowchart-based documentation for C++ codes. Comput Phys Commun 196:497–505. https://doi.org/10.1016/j.cpc.2015.05.029
33. Flater D, Martin P, Crane M (2009) Rendering UML Activity Diagrams as Human-Readable Text. *Ike*, pp 207–213, [Online]. Available: http://dblp.uni-trier.de/db/conf/ike/ike2009.html#FlaterMC09
34. Riesco M, Fondón MD, Álvarez D (2008) Using graphviz as a low-cost option to facilitate the understanding of unix process system calls. Electron Notes Theor Comput Sci 224(2):89–95. https://doi.org/10.1016/j.entcs.2008.12.052
35. Kraft NA, Lloyd EL, Malloy BA, Clarke PJ (2006) The implementation of an extensible system for comparison and visualization of class ordering methodologies. J Syst Softw 79(8):1092–1109. https://doi.org/10.1016/j.jss.2005.10.019
36. Haw SC, Rao GSVRK (2007) A comparative study and benchmarking on XML parsers. Int Conf Adv Commun Technol ICACT 1:321–325. https://doi.org/10.1109/ICACT.2007.358364
37. Oliveira B, Santos V, Belo O (2013) Processing XML with Java—A Performance Benchmark. Int J New Comput Architect Appl 3(1):72–85
38. "UML sequence diagrams overview of graphical notation - lifeline, message, execution specification, interaction use, etc." https://www.uml-diagrams.org/sequence-diagrams.html Accessed Mar 05 2018

39. Geary RF, Rahman N, Raman R, Raman V (2006) A simple optimal representation for balanced parentheses. Theoret Comput Sci 368(3):231–246. https://doi.org/10.1016/j.tcs.2006.09.014
40. Giuseppe N (2018) Template-based code generation with apache velocity, Part 1 - O'Reilly Media. http://www.onjava.com/pub/a/onjava/2004/05/05/cg-vel1.html Accessed Mar 05 2018
41. Li Z, Jiang Y, Zhang XJ, Xu HY (2020) The metric for automatic code generation. Procedia Comput Sci 166:279–286. https://doi.org/10.1016/j.procs.2020.02.099