Automatic Code Generation From UML Behavioural Models

A Thesis submitted to

Cochin University of Science and Technology

in partial fulfilment of the requirements

for the award of the degree of

Doctor of Philosophy

Under the Faculty of Technology

By

SUNITHA E V

(Reg No. 4213)

Under the Guidance of

Dr. PHILIP SAMUEL

Department of Computer Science

Cochin University of Science and Technology

Kochi - 682 022, Kerala, India

April 2019

Automatic Code Generation From UML Behavioural Models

Ph.D. Thesis under the Faculty of Technology

Author

SUNITHA E V

Research Scholar
Department of Computer Science
Cochin University of Science and Technology
Kochi - 682022
Email: sunithaev@gmail.com

Supervising Guide

Dr PHILIP SAMUEL

Professor
Department of Computer Science
Cochin University of Science and Technology
Kochi - 682022
Email: philipcusat@gmail.com

Department of Computer Science

Cochin University of Science and Technology

Kochi -682022

CERTIFICATE

This is to certify that the thesis entitled "**Automatic Code Generation From UML Behavioural Models**" submitted by **Ms. Sunitha E.V.** (Reg. No. 4213) to the Cochin University of Science and Technology, Kochi for the award of the degree of Doctor of Philosophy is a bonafide record of research work carried out by her under my supervision and guidance in the Department of Computer Science, Cochin University of Science and Technology. The content of this thesis, in full or in parts, have not been submitted to any other University or Institute for the award of any degree or diploma. I further certify that the corrections and modifications suggested by the audience during the pre-synopsis seminar and recommended by the Doctoral Committee of Ms. Sunitha E V are incorporated in the thesis.

Kochi
25th April 2019

Dr. Philip Samuel
(Supervising Guide)

DECLARATION

I hereby declare that the work presented in this thesis entitled "**Automatic Code Generation From UML Behavioural Models**" is based on the original work done by me under the guidance of Dr. Philip Samuel, Professor, Department of Computer Science, Cochin University of Science and Technology and has not been included in any other thesis submitted previously for the award of any degree.

Kochi **Sunitha E V**

25th April 2019

ACKNOWLEDGMENTS

At the outset, I thank **God Almighty** for providing me the great opportunity to do this research work and complete it within the stipulated time.

I have great pleasure in expressing my profound gratitude to my guide, **Dr. Philip Samuel**, for the motivation and guidance provided throughout this research work. I have been extremely lucky to have a guide who cared so much about my work and who responded to my questions and queries so promptly. I appreciate his approachability and the trust bestowed on my research efforts. Consistent support, patient hearing and sincere encouragement provided throughout the period are gratefully acknowledged.

I wish to express my sincere gratitude to **Dr. G. Santhosh Kumar**, Head, Department of Computer Science, Cochin University of Science and Technology, for his guidance, help and encouragement throughout the period of this work. I extent my deep sense of gratitude to **Prof. (Dr.) Sumam Mary Idicula** for her support as the Co-guide during the initial stage of the research work. I am grateful to **Dr. K. Poulose Jacob**, ex Pro-Vice-Chancellor of Cochin University of Science and Technology, for being a source of support and encouragement. His sincerity and supportive attitude enabled the successful completion of this work.

I thank the doctoral committee members, all the faculty members of Department of Computer Science and Information Technology Division, School of Engineering, Cochin University of Science and Technology for the cooperation extended towards me. The excellent computing and library facilities offered by the department were really supportive for carrying out my thesis work. The support of all the non teaching staff of Department of

Computer Science, Cochin University of Science and Technology throughout the period is gratefully acknowledged.

I have no words to express my deep sense of gratitude to my colleagues and well wishers at Toc H Institute of Science and Technology, Arakkunnam, for helping me to complete this work. Special thanks to **Dr. K Varghese**, Founder Director & Manager, other management members and Principal, TIST for their valuable cooperation and encouragement during this period. My sincere thanks to all teaching & non teaching staff of department of Information Technology TIST for their cordial relations, sincere co-operation and valuable help during different stages of this research work.

I would like to extend my sincere thanks to my friends **Ms. Rani Augustine, Ms. Lichia Reghu, Mr. Ajeesh G Krishnan** and **Dr. Joby Jose** for their valuable and timely support for the thesis preparation.

I am blessed to have the unconditional everlasting love from my parents **Late Mr. E N Viswanathan** & **Mrs. Lalitha Viswanathan** and my parents in law **Mr. T K Mohanan** & **Mrs. Lalitha Mohanan** whose encouragement and support always kept me overcoming hard times. I dedicate this accomplishment to them.

Finally, but most importantly, I would like to thank my dearest husband, **Prof. Arun Mohan**, for being so understanding and for putting up with me through the toughest moments of my life. I thank God for enlightening my life with his presence. Without his sacrifices, love and support, this thesis would not have been possible. I also dedicate this Ph.D. thesis to my two lovely children, **Meenakshi** and **Parvathi** who are the pride and joy of my life.

Sunitha E V

CONTENTS

ABSTRACT

The emergence of Unified Modeling Language (UML) as the industrial standard for modeling software systems has encouraged the use of automated tools that facilitate the development process from analysis through coding. Software models are important in building large software systems. Even though these models are used to simplify the software system, they can be in themselves, quite complicated. It is not all clear how to build these software from the models in the best way. This work tackles that problem. In UML, the static structure of a system is represented by a class diagram while the dynamic behavior of the system is represented by a set of behavioural diagrams. To facilitate the software development process, it would be ideal to have tools that automatically generate or help to generate source code from the models. In this thesis, we present a novel approach to automatically generate source code from the UML behavioural models. An object oriented approach has been proposed to generate implementation code from use case diagram, sequence diagram, activity diagram and state chart diagram in an object-oriented programming language.

The functional requirements of a software system expressed in use case models are utilized to modularize the source code. The object interactions to accomplish the use cases have been converted to the method declarations and definition statements. It is useful to fine tune the user requirements. The process flow depicted using the activity models are considered to update the method definition of the system. The additional information regarding the pre and post conditions of each activity, the method parameters, etc., are incorporated in the activity diagram using the Object Constraint Language. Sequence diagrams can be used in association with the activity nodes to include the object interaction details. These interactions are made use of to complete the method definitions. A new design pattern to implement state chart diagram with hierarchical, concurrent and history states is proposed in this thesis. These methods have been implemented by developing an automatic code generator which converts the UML behavioural models to the source code. It has been used for the evaluation of the methods. Our approach successfully generated code from UML behaviour models. The research findings conclude that the automatic code generation from the system models reduces the software development efforts and time.

GLOSSARY OF SYMBOLS AND ABBREVIATIONS

AD	:	Activity Diagram
$\mathcal{AG}_{ocl}$	:	Activity Graph enhanced with OCL
$\mathcal{AG}_{sd}$	:	Activity Graph enhanced with Sequence Diagram
AM	:	Activity Model
AN	:	Activity Node
$\mathcal{AN}_{d}$	:	Action Node
BPEL4WS	:	Business Process Execution Language for Web Services
CASE	:	Computer Aided Software Engineering
$\mathcal{CBA}$	:	Call Behavior Action
CD	:	Class Diagram
CM	:	Collaboration Model
CN	:	Control Node
CPNs	:	Colored Petri Nets
DTD	:	Document Type Definition
DSML	:	Domain Specific Modeling Language
FSM	:	Finite State Machine
HSM	:	Hierarchical State Machine
HST	:	Hierarchical Structure Tree
IDE	:	Integrated Development Environment
MDA	:	Model Driven Architecture
MDD	:	Model Driven Development
OCL	:	Object Constraint Language

OMG	:	Object Management Group
OMT	:	Object Modeling Technique
OO	:	Object Oriented
OOD	:	Object Oriented Development
OOSE	:	Object Oriented Software Engineering
PIM	:	Platform Independent Model
PSM	:	Platform Specific Model
QHSM	:	Quantum Hierarchical State Machine
rCOS	:	Relational Calculus of Object Systems
RSL	:	RAIS Specification Language
RTPA	:	Real Time Process Algebra
RTPA-VM	:	Real Time Process Algebra - virtual machine
SC	:	Source Code
SD	:	Sequence Diagram
SIG	:	Sequence Interactions Graph
SRS	:	Software Requirement Specification
UI	:	User Interface
UML	:	Unified Modeling Language
UML-RSDS	:	UML Reactive System Development Support
XMI	:	XML Metadata Interchange
XML	:	eXtensible Markup Language
XSL	:	eXtensible Stylesheet Language
XSLT	:	eXtensible Stylesheet Language Transformations

LIST OF FIGURES

LIST OF TABLES

Chapter 1 INTRODUCTION

1.1 Introduction

Software systems are vital part of our day to day life. Automobiles, mobile phones, computers, TV and many other home appliances depend on software. While the demand of software is increasing, its complexity increases exponentially. Eventually, the process of building the software becomes harder and harder to manage and much more difficult to maintain.

In order to make the software development process easier and manageable, the engineers started using different diagrams to design the system. Drawings can easily convey information than words. Unified Modeling Language (UML) is the best choice for software system design [99]. Now-a-days UML is widely accepted as a modeling language for software systems.

The use of UML eases the software designing process. Once the design is ready, the next phase is software implementation. How the system design in UML can aid the implementation phase, so that the complexity of the coding phase gets reduced? Is it possible to generate source code from the UML models? Yes. This is one interesting solution to reduce the complexity of software development process, that is, generating source code automatically from the software system designs [4, 107]. Software designs get an upper hand in software development process and they remain as the leading element in the process. It is called as Model Driven Development (MDD).

1.2 Unified Modeling Language

UML is the de-facto standard in industry for designing software systems [139, 103]. As the complexity of the software is increased, the lines of code and the interfaces to other software are also increased drastically. When this complexity became unmanageable, researchers in the area of Object Oriented (OO) development started proposing visualizing techniques to present the system design. Grady Booch, Ivar Jacobson and James Rumbaugh were the three most important contributors in this field. Booch method proposed by Grady Booch, Object Oriented Software Engineering (OOSE) method contributed by Ivar Jacobson and Object Modeling Technique (OMT) by James Rumbaugh [55].

Another big movement in this field was the unification of these three methods. They unified their own approaches to software modelling in the rules and definitions of UML, today being the standard for building object-oriented software systems. The software developers started using UML notations and the CASE tool vendors started supporting UML in their tools. The Object Management Group (OMG) maintains a list of available tools helping the software engineer using UML and this list reflects the high reputation of UML as a modelling language.

UML provides, mainly, eight basic diagrams [99]. The static structure of a software system is modeled using Class diagram. It describes the static aspect of the system in the form of classes, packages and their relations. The functionality of the software system is described using the Use case diagram, from the viewpoint of the user. It gives a highly abstract view of the system. The interaction between objects to accomplish a use case is depicted using Sequence Diagram. The interactions as well as the structural relationships of the objects are designed using Collaboration Diagrams. The entire life cycle of an object, which includes the different states of the object, the state transitions and the events that force these state changes are illustrated using the State Chart Diagram. The sequence of activities involved in a use case realization and the different objects involved in it are described using the Activity Diagram. Dependency between different components in a system is designed using Component Diagram. Finally, Deployment Diagram gives the architecture of the system.

The entire software development process is supported by UML that is, starting from analysis till the maintenance of the system. It helps forward engineering as well as reverse engineering. That is the source code can be generated from the UML designs and the systems designs can be generated from the source code which will be more helpful during the maintenance phase. The UML designs give us the high level design details of the system.

When a programmer implements these designs, he/she includes so many implementation specific details [87] like, variable declarations, initializations, pre-defined constant values, method definitions, class definitions, etc. Automatic code generation could be possible in its real sense, only when we are able to automatically generate all these implementation specific details. In forward engineering the completeness of the code generated is the main issue. Therefore, how to develop UML based environment for software development is a hot research issue.

1.3 Automatic Code Generation From UML Behavioural Models

The structure and behaviour of the system is modeled using UML diagrams. UML class diagrams are used for system structure modeling and it allows source code generation [97]. It gives the details about the problem domain unit, its characteristics and operations. Class declarations with attributes and method signatures can be generated from the UML class diagram [29]. It gives the structure of the software system. It cannot be executed since the generated code is incomplete. The developers have to explicitly complete the source code with the object behaviours and interactions. Then the system becomes executable. An alternative for this is the code generation from the UML behavioural models of the system. Use Case Diagram, Sequence Diagram, Activity Diagram and State chart Diagram are the main UML behavioral models.

Use case diagrams represent the different functionalities of the software system and so it is used for modularizing the source code (implementation code) of the system based on the features it supports. Each use case in a use-case diagram represents a functional service of the system that is to be used by a specific actor and satisfies a requirement specified in terms of a pair of pre and post conditions. That is, it gives the external objects, which interact with the system. The features that are accessible by the different external objects, the extended or included operations of each functionality etc are described in the use case diagram. These details are used for code generation, to modularize the code, to provide access privileges to the objects and to organize internal function calls.

Sequence diagrams show the interaction between objects with the sequence number. It gives the messages passed (the function calls) between objects to accomplish a use case. In addition to these details, it also gives some details indirectly. Each message to an object says that the class of the object should have methods to handle those messages. These details will help us to generate method definitions as well as to update the class diagrams.

Activity diagrams give the process flow. It describes different activities in a process. Activity diagram can be used in design to model different parts of the system. It can be used to design the entire process flow of the system. In a deeper view, we can draw activity diagram corresponds to each use case in the use case diagram. Sub activity diagrams can be drawn to expand a complex activity in the main activity diagram. For each activity, there can be pre and post conditions. Concurrent activities, decision making etc., can be modeled in the activity diagram. So, it helps us to generate the source for the main function of the software system. It can also be used for generating method definitions. The constraints, decision making statements etc can be generated from the activity diagram.

The state chart diagrams of UML can be used to automatically generate program source code. Code generation from state charts diagrams only generates the behavior code for a particular object. It generates code for one class only with which the state chart is attached. The developer has to explicitly join this code with other parts of the application to make the code for the entire application.

1.4 Motivation

Automatic code generation from system designs is an emerging research area [3, 73, 88, 115, 33, 87, 35]. This concept has versatile dimensions in software industry. The idea behind this concept is that, before implementing a system, we can model it using standard notations, like UML [62, 61, 9, 48]. Then automatically generate code from these models. This idea is quite interesting since the coding and testing phases of software development process are very much expensive. It can reduce the effort we put for coding and testing and in turn, can improve the quality of the software.

Due to evolution, adaptation and changing requirements, software maintenance is a challenging task. During maintenance phase, the maintenance engineer changes the source code, but not the designs. So, each maintenance work reduces the correlation with the design and the source code. Gradually the system design becomes obsolete and it may not have any relation with the actual system [115]. The model based code generation gives a solution for this scenario. During maintenance the engineer can change the system models instead of the actual code and then generate code out of the system model. The idea is pretty good, but the implementation is difficult.

UML supports object orientation in the design phase. UML help us to design the structure as well as behavior of the system. Similarly, OO programming languages like Java, C++, C# etc., are useful in the implementation phase. This helps us to continue the object orientation in the design phase to the implementation phase. There are some elements in UML design which can be directly mapped to any object oriented programming construct. Some elements in UML cannot be directly mapped to any programming element. Earlier, the designers design the system models using UML or other tools and hand it over to the software engineers for coding. The software engineers had to start from the scratch, beginning from the inclusion of header files, declaration of variables etc. Over time, this scenario had been changed and there came some CASE tools, IDEs etc., for supporting the software engineers. These tools generate skeletal code from the designs we have modeled in UML or similar languages so that the programmer need not start from the scratch.

In the next generation of software development, there comes the Model Driven Development (MDD) [114, 109, 129, 144]. MDD describes methods to develop software purely based on the system design. Even though the code generation from the UML models sounds an interesting concept, it is not an easy one to implement. The research in this area starts with the code generation from the structural models like class diagrams [29, 139, 146, 147].

The system design may include class diagrams, state charts, activity diagrams, sequence diagrams etc. Some methods are available to convert UML Class Diagram to source code. The OO languages support the class concept. The class declaration statements, class definition statements, method definition statements, object creation, method invocation statement etc., are available in the existing OO languages.

A method to convert the class diagram represented in XMI format to Java code is presented in Bjoraa [29]. They have developed a prototype to output one Java file per class specified in the class diagram. The class diagram drawn in UML will be converted to XMI format. The XMI file will be parsed using XML parser and extracts the details, like class name, attributes and methods. Using this information the skeleton of the class definition will be produced in Java. [139] uses stereotyped class diagram for code generation. Classes marked with the stereotype <<entity>> will be mapped into an interface and a pair of implementing classes. One class will be abstract class and the other one will be instantiable.

Later OCL expressions are added to UML to specify constraints. The conversion of OCL enhanced class diagrams to some specification languages, like RAIS Specification Language (RSL), or implementation code in C++, Java etc., has been proposed by some researchers [91, 52, 70, 148, 149].

UML Reactive System Development Support (UML-**RSDS**) is another subset of UML [67, 74, 72]. It provides semantics for class, use case diagram and OCL for automatic code generation. OCL constraints are used to show the relations between system models. In this approach OCL is used for specifying class and state invariants, pre and post conditions for operations and use cases, etc.

Anyhow, these methods are not capable of converting UML behavioural models to source code. Behavioural models like state chart diagram, activity diagram etc., cannot be directly mapped to OO program [56]. This is because of the lack of programming elements that can represent the elements in these diagrams. In addition to that, the code generated from the structural models will have a skeletal code, not the complete one, since the system behavior is not taken into account. So, the studies in code generation diverts to behavioral models like activity diagram, sequence diagram, state diagram etc.

The literature in the area of code generation says that the prime issue in the code generation is the gap between the model and the software system. A model-system gap exists primarily due to the different levels of abstraction. Software designs are used to communicate with the clients. So it cannot be implementation oriented. It should clearly explain how the client requirements will be satisfied by the software system. So the designs are at high level of abstraction. For example, Use case diagram just gives the different functional services provided by the system. Sequence diagram gives the object interactions; the messages passed between objects are not so relevant for the client. If we include more implementation specific details in the design, it will be complex for the clients to understand.

The object-oriented methodologies describe the steps to be followed during the analysis and design phase, but fail to describe how the analysis and design models of a system shall be converted into implementation code. A big problem in the development of a system through object-oriented methodologies is that, even after having created good models, it is difficult for a large fraction of software developers to convert the design models into source code. It would be ideal to have tools that support the developer and automatically generate or help to generate source code from the models. In this thesis we address these difficulties to automatically generate source code from the object-oriented system designs.

1.5 Problem Statement

The goal of this research is to investigate methods to automatically generate code from the UML behavioural models.

1.6 Research Objectives

The aim of this research is to device methods to automatically generate code from the UML behavioural models. It includes approaches to automatically generate source code from the UML Use Case diagram, Sequence Diagram, Activity Diagram and State Chart Diagram. Thus the objectives addressed in this research work are:

- To device a method to generate code from UML Use Case Diagram.
- To device a method to generate code from UML Sequence Diagram.
- To device a method to generate code from UML Activity Diagram.
- To device a method to generate code from UML State Chart Diagram.

1.7 Thesis Overview

The rest of the thesis is organized into 7 chapters.

The **Chapter 2** gives a systematic literature review on the existing code generation methods based on the behavioural models; use-case diagram, sequence diagram, activity diagram and state chart diagram. Code generation methods described in the literature review have its own advantages and limitations. The diagrams used for system design depends on the kind of software we are going to develop. The percentage of code generated in each method varies depends on the features that are considered for the code generation. Algorithms for code generation is lacking in the existing literatures. Moreover, some features of the diagrams are not considered for code generation in these methods. The proposed code generation methods in this thesis addressed the features that are not explored in the existing works and there by generates more code than the existing methods. In addition to that, theoretical proof for connecting different system modeling diagrams and precise steps for code generation is proposed in this thesis. The formal semantics for enhancing the UML activity diagram with OCL, its proof and the algorithm for code generation from the OCL enhanced activity diagram are presented in this thesis.

The **chapter 3** presented different UML behavioral diagrams and their use in automatic code generation. This chapter gives an overview of the UML use-case diagram, sequence diagram, activity diagram and state chart diagram and the contribution of these diagrams in the code generation.

The **chapter 4** discussed the code generation from the use-case diagrams and sequence diagrams. Use-case diagram is used to frame the context class and the sequence diagram is used to add details to the class. The code generation from use case diagram is done in five steps and from sequence diagram is done in three steps. Algorithm for each one is given in the chapter. The algorithms give a formal way to do the prototype generation and this method is easy to implement. The analysis of the proposed method shows that it can generate even more than 30% of code for frequently interacted classes. This is a promising result in code generation from the use case models.

The **chapter 5** discussed the cod generation from the OCL enhanced activity models. A Theoretical proof to connect OCL statements for operation body, actual parameters, initial values, instances and guard conditions with the activity diagram is depicted here. The operational semantics for OCL enhanced activity diagram and a concrete method for converting it into source code are also explained in this chapter. The proposed algorithms give a proper guideline for the code generation from OCL enhanced activity diagram. ActivityOCLKode, the tool implemented based on proposed algorithm, provides a user friendly environment for the users to model the process flow based software systems. The evaluation of the tool shows the proposed method of code generation helps us to generate more than 83% code. When the OCL is added with the activity diagrams, this raises up to 84.4%. The code, generated from OCL, is very crucial since it includes method definitions and the specific pre- and post conditions. Moreover, the time required for code generation based on the proposed method is 11.46 milliseconds approximately. The use of OCL improves the percentage of code generated. The use of XML to save the models in text format improves the portability of the models. UML models, OCL and XML all are widely accepted and used in software industry and so the proposed method can be easily adapted to the software development process in the software industry.

The **chapter 6** proposed a method to combine activity models and sequence models to improve the code generation. Activity diagram alone cannot give the implementation details like object interactions. We found a formal association between activity and sequence diagrams to add object interaction details to the work flow. Moreover, we formulated an algorithm, *Am_To_Prototype*, which is composed of two subroutines named *Method_Body* & *Excecution_Logic*, to generate code from the combined model of activity and sequence diagrams consisting of concurrent activities. The authors compared the proposed method with other research outcomes with respect to workflow automation, support for concurrency, etc. The proposed algorithms are able to generate class definition, method definition and control flow.

The **chapter 7** proposed a method for automatic code generation from UML state chart diagrams. The event driven systems can be modeled and implemented using UML state chart diagrams. The existing programming elements cannot effectively implement two main components of the state diagram namely state hierarchy and concurrency. We proposed a novel design pattern for the implementation of the state diagram which includes hierarchical, concurrent and history states. The state transitions of parallel states are delegated to the composite state class. The architecture of the code generator and the step by step process of code generation from UML state machine are proposed in the chapter.

The **chapter 8** concludes the thesis by presenting the main contributions and future research directions.

Chapter 2 LITERATURE REVIEW

2.1 Introduction

Design and coding are the two important phases of software development process. Designers draw the structural and behavioral models of the software system according to the analysts' report and Software Requirement Specification (SRS). Nowadays, use of UML [99] to design the models of a system is very common. Programmers develop the implementation code based on the design models and the SRS. Each module of the whole system will be given to different programmers. Manual programming is very expensive and error prone.

In addition, some programmers may not add proper documentation in the source code. This reduces the readability and understandability of the code and thereby making the maintenance of the software very difficult. Syntax errors are another unavoidable headache in manual coding. In short, a lion's share of the software development effort is put on coding and debugging [108]. A better solution to this problem is the use of automatic code generators.

UML is one of the designing languages which support object orientation in the design phase. It supports the important concepts of Object Oriented Development (OOD) such as, abstraction, inheritance, modularity, polymorphism etc. UML help us to design the structure as well as behavior of the system. They are called structural modeling and behavioral modeling. Structure diagram includes class diagram, object diagram, deployment diagram etc. Behavioral diagrams include activity diagram, state chart diagram, sequence diagram etc.

The code generated from the structural models will have a skeletal code, not the complete one, since the system behavior is not taken into account. So, the studies in code generation diverts to behavioral models like activity diagram, sequence diagram, state diagram etc. In this chapter we discuss code generators and different approaches for code generation from UML models.

2.2 Code Generators

According to the software engineering practices, major share of the software development effort is put on manual coding and debugging [108]. In this scenario, CASE tools with automatic code generators can do wonders. The code generators can also be called as model compilers [11, 40, 87], which take UML models as input and produce implementation code as output. It can do model validation too. Code generators separate the system model from the source code and thereby reduce the complexity of the software development. It helps us to save time by generating a major part of the source code. The code generators handle the code duplication and refactoring. Moreover, the code generators help us to impose coding standards and so the quality of the source code will be improved.

Code generators use eXtensible Stylesheet Language (XSL) [20] for representing system models which takes the advantage of widely accepted XML standard. It gives a standard and flexible data format. The improved performance over code generation and maintenance obtained by using a code generator does comes at the expense of the effort to create an information model and to customize the code generation logic. The benefits of code generation outweigh the additional overhead, especially in larger projects.

Georgescu [20] categorizes the code generators into two types. First one is programmer centric and the second one is designer centric.

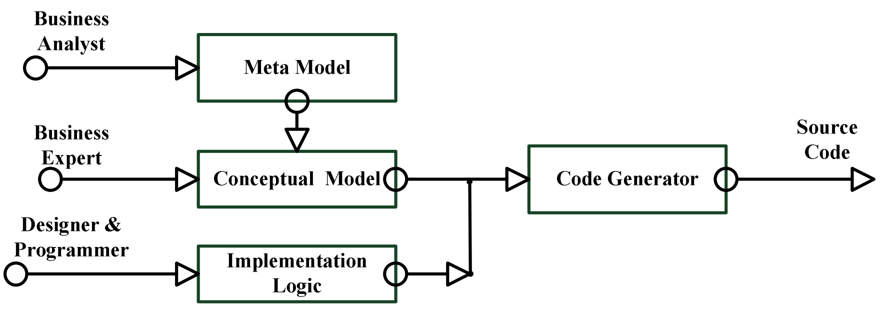


Figure 2.1 : Programmer centric code generator

Figure 2.1 shows the components of a programmer centric code generator. It takes two inputs, first one is a conceptual model and the second one is the implementation logic. Business expert provides the information models or data which explain the conceptual model of the problem domain. The additional data required here is the meta model, which contains the data about the model. System analyst will provide this information. Programmer plays an important role in this type of code generators. The key parts of the code generation will be done based on the input of the programmer not from the conceptual model of the system. That is the implementation logic is given by the programmer with the help of the designer of the system. It specifies the instructions to generate code.

A second type of code generator is model centric code generators. It has a more elaborated architecture that involves the creation of an intermediate design model and possibly iterating through several design models before the final step of code generation (see Figure 2.2). Along with the conceptual model, the designer adds the design logic to generate the system model. Designers play an important role in this type of code generators. The major part of the code generation logic is given by the designer through the design logic and hence the name.

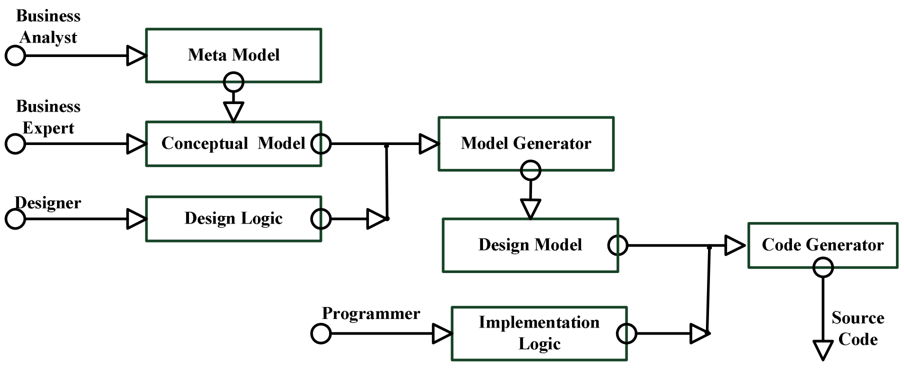


Figure 2.2: Designer centric code generator

We use the designer centric code generation approach in our research work. UML is used as the software system modeling tool. The code generation from UML diagrams is an ever growing research area. Many authors contributed to this [28, 29, 39, 49, 53, 70, 77, 84, 85, 90, 92, 93, 107, 110, 128, 132, 133, 139, 141, 142, 143] even from late 90s.

Other than standard modeling tools like UML, some research works proposed code generation from formal specification of the system [21], or code generation from new modeling languages [45].

Key problems in automatic code generation on the basis of formal methods are that: (a) Formal specifications are abstract descriptions of a system and each specification may be satisfied by many different implementations. (b) Most formal notations use mathematics to express systems behaviors. There is no direct transformability for some of the abstract descriptions in executable target languages. Cyprian [21] presented a method to transform the formal specifications in Real Time Process Algebra (RTPA) to Java code. RTPA denotes system behaviors by meta and complex processes.

Most RTPA processes can be translated directly into Java. For those processes that cannot be translated into RTPA directly, a special class in the virtual machine (RTPA-VM) will be called, which provides a set of predefined functions for executing the nontraditional processes.

Code generation from new modeling language, ThingML, is proposed by Harrand [45] to utilize the benefits of the Model Based System Engineering (MBSE). ThingML is a domain specific modeling language (DSML). It supports automated code generation from system models, thereby increases the productivity of the software development team. [45] states that they implement the behavior of a system using code generation from the state machines of the system. For code generation, they use existing design patterns in C++ or Java, or else other existing frameworks such as State.js [95]. They focus on heterogeneous target platforms.

In the following sections we present the existing methods for code generation from the UML use case models, sequence models, activity models and state chart models.

2.3 Existing Methods For Code Generation From Use Case Diagrams And Sequence Diagrams

UML use-case models are used primarily for the requirement analysis in software development [38]. It draws the external view of a software system. It describes the functionalities of the system which are used or initiated by a human user. A use-case description is associated with each use-case in the use-case diagram which explains the name of the use-case, the summary of its working, the actor (human user) who uses the use-case, the pre and post conditions, the description of the use-case and the alternative options in the use-case. All these details will be given in natural languages like English [116].

The use case model is used throughout the software development. In the requirement specification phase, it is used for specifying functional requirements. This will be used in analysis and design phase as the base input. Moreover, the use case model is used as input to iteration planning, for test case generation and as a major component for user documentation.

UML use case and sequence diagrams can be used to generate source code of a software system. The use cases give the list of services provided by the system. The sequence diagrams allow us to expand the service methods as a sequence of method calls.

Prototyping is an efficient and effective way to close the gap between customers and designers in their understanding of the system and its requirements and validating the customers' requirements. Li [141] define system requirements model as a pair of a conceptual class model and a use-case model. They decompose each use case declared with its pre and post conditions into a sequence of primitive actions and then generate an executable source code in Java. The prototype can be executed for validating the use cases under the given conceptual class model and checking the consistency of the requirements model.

For complex use cases, we need to draw their corresponding sequence diagrams. There are a couple of methods to generate code from such sequence diagrams [86, 22, 30, 102, 60, 109, 107, 83, 142].

Sequence diagram along with class diagram help us to generate prototype of the system. UML class diagram is used to generate a structural view and sequence diagrams to generate the behavior view [86, 102]. The system design contains a main sequence diagram which defines main method and defines the start point of the behavioral code. Structural code is generated from the class diagram. The sequence of methods is captured including returns and arguments, from the sequence diagrams.

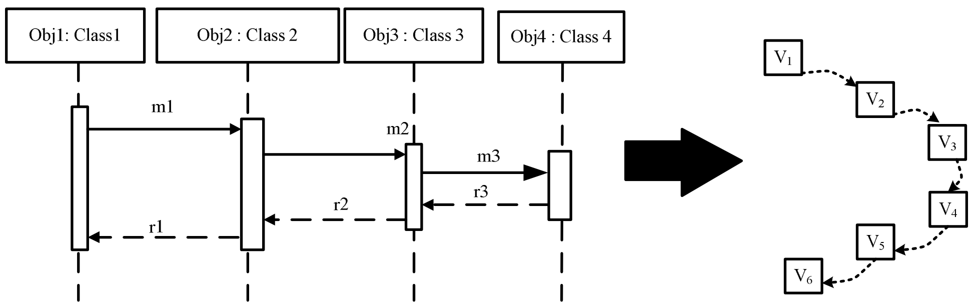


Figure 2.3: SD and corresponding SIG

Another approach for code generation from sequence diagram is based on intermediate models [22]. The sequence diagrams (SD) first converted to sequence interactions graphs (SIG) with help of a set of mapping rules. These graphs contain information like messages, control flow and method scope of interactions. This information is then used to generate code. During code generation, first identify the subgraphs of the graph model which belongs to the same method scope of a class method. Then apply the mapping rules to the model elements contained in the subgraphs to generate the code of different class methods.

Figure 2.3 gives a sequence diagram and its corresponding SIG. The SIG contains 6 nodes V_1, V_2, V_3, V_4, V_5 and V_6 corresponding to the 6 messages (m1, m2, m3, r3, r2 and r1), respectively.

A reverse approach is presented by Aziz [93] where the sequence diagrams are generated from the source code. This reverse engineering helps us to extract system abstractions and design information from existing software.

Instead of SIG, an intermediate structural model representing the Java platform specific model (PSM) can be generated from the sequence diagram of system's internal behavior, which is a platform independent model (PIM) [30]. A set of model transformation rules has been defined for the same. Objects involved in use cases and sequence diagrams are transformed to Java classes by merging details in the domain class diagram. Then, the methods involved in the sequence diagram interaction are converted to source code. Method body has to be added explicitly since it's not available from the sequence diagram.

A different approach is proposed by [60]. During requirements engineering, each use case is elaborated with sequence diagrams. These sequence diagrams are combined into one to form a global single sequence diagram capturing the behavior of the entire system. The use case diagram and all sequence diagrams are transformed into Hierarchical Colored Petri Nets (CPNs). Finally, a system prototype and code is generated from the global single sequence diagram and can be embedded in a user interface (UI) builder environment for further refinement.

Ruben Campos has proposed a method for xUML engine which hide the details behind translating UML models into a high level program [109]. The sequence diagram is selected as the focal point of execution in that xUML Engine. It uses the class diagram as the entry point in implementing the class methods and the Activity diagrams are used to implement the details of a class method.

Instead of intermediate models, intermediate languages can also be used. In [107], Relational Calculus of Object Systems (rCOS) is used as the intermediate language. The sequence diagrams and the class diagram are first checked for consistency. Error report is generated if there is any inconsistency. Otherwise the diagrams are given for code generation.

During the generation of method bodies, traverse through the sequence diagram. When a message is sent (or in other words, method is called), the signature of the method is created. For a send point, if it calls method m, the algorithm writes the signature of m to the body of the method that is currently being generated, leaving the method body unfinished, begins to write the body of method m.

Thongmak [83] converts the sequence diagrams to java code. They defined transformation rules for the same. According to their method, class diagram and sequence diagrams are transformed to a meta-model. This meta-model will be then converted to java code using the transformation rules. The programmer's intervention is required to complete the program.

2.4 Existing Methods For Code Generation From Activity Diagrams

Behavioral modeling is very much important in the context of automatic code generation, since it helps us to represent the control flow in the system. Activity diagram is one of the most important diagrams for behavioral modeling. It is the only UML diagram which models control flow (work flow). Activity diagram gives the activity model of the system which shows the workflow from activity to activity. Activity diagrams are activity centric and it shows flow of control from activity to activity.

Activity diagram can show the group of activities done by different objects in the system. We can specify which object is responsible for which activity. This is a unique feature of activity diagram compared with other behavior diagrams like state chart diagrams.

An approach to the model driven generation of programs in the Business Process Execution Language for Web Services (BPEL4WS) which transforms a platform independent model to platform specific model is described in Koehler [53]. Business process modeling is done using the activity diagrams. They define rules for integrating business process. This rule helps them to reduce complex activity diagrams to comparatively simple diagrams which do not contain loops. According to their approach, the control flow models will be analyzed first. Sub processes in the model will be identified.

These are the regions in the model which have a single entry node to the region and single exit node from the region. Check whether this region can be reduced to a single node. To find the reducibility they provide some rules. Further, they provide a declarative method to convert these reduced models to BPEL4WS.

Business process is modeled using UML2.0 activity diagram in Yin [143]. The semantics of it is also given by [143]. Set of the activities, including the Primitive Actions, CallBehaviorActions and Pseudo actions; set of transitions, flow relation; InitialNode and ActivityFinalNode; set of local variables within the activity diagram are included in the formal semantics of the business process model. A PrimitiveAction defines a method that is to be called by its actor to perform its functionality. The functionality is specified by a pair of OCL precondition and postcondition, which describes a relation between the states of the system before and after the execution of the PrimitveAction. It is implemented as a sequence of atomic actions simulating the state change. Method interactions are not considered here.

An activity within an activity diagram is specified by a pair of precondition and postcondition in OCL [63]. The objects declared in the class diagram are used in the OCL expressions to carry out functionalities of the activities. By analyzing the semantics of the precondition and postcondition, the behavior of an activity can be generally decomposed into a sequence of atomic actions manipulating the objects. An activity is transformed to a Java class with the sequence of atomic actions. An activity diagram is transformed into a Java class with a method simulating the execution of it. Activity classes are instantiated to perform their functionalities according to the control flow defined in the activity diagram. Pins used as arguments of activities are transformed into parameters of the calling of the activity classes.

The Object Management Group (OMG) has specified a subset of UML 2.0 exclusively for Model Driven Development [129]. This subset is named as Foundational UML (fUML) [129]. fUML considers only class diagram and activity diagram. In order to improve the precision an action language named Alf [96] is used with fUML. It is a textual action language. However, it does not have any advantage over UML and OCL, since fUML requires detailed modeling and precisions should be added using an action language like OCL. A sound knowledge in fUML and Alf is necessary to convert fUML models to code. The same thing can be done with UML and

OCL and with less effort since most of the developers and designers are familiar with those standards.

2.5 Existing Methods For Code Generation From State Chart Diagrams

UML state chart diagrams can effectively represent the behavior of event driven systems aka reactive systems. The behavior of the event-driven system changes with the interactions (events) with the environment. The state diagrams show that the behavior of a system depends on the current input to the system as well as the previous interactions by the environment. Event driven systems modeled using the state machines can represent the full life cycle of an object. The different states of the object and the transition between those states are all portrayed in this. The challenge is to work out an efficient method to convert state charts to a program since there is no programming construct exist to directly represent elements in the state diagram.

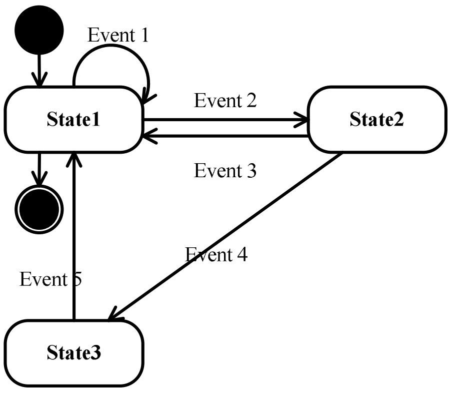


Figure 2.4: Sample state chart diagram

Dominguez [26] presented a review of research works that propose methods to implement UML state chart diagrams. Dominguez summarizes the review by saying that the state transition process in most of the works is based on switch statement, state table or state design patterns. Another key finding of [26] is that very few papers support hierarchy and concurrency of states. State machine implementation techniques include nested switch statement, state table and state design patterns.

2.5.1 State machine implementation using switch statement:

Using switch statement the system state is implemented as a variable and events are implemented as methods. The general structure of the state chart implementation using switch statement for the Figure 2.4 is shown below.

```
switch(CurrentState)
{
case State1 :
          switch(NewEvent)
          { case ‘Event2’:NextState = State2; break;
            case ‘Event1’: NextState = State1; break;
          }
          break;
case State2 :
          switch(NewEvent)
          { case ‘Event4’:NextState = State3; break;
            case ‘Event3’:NextState = State1; break;
          }
          break;
case State3 :
          switch(NewEvent)
          {  case ‘Event5’:NextState = State1; break;
          }
          break;
}
```

The switch statement receives the current state and the nested switch statement chooses appropriate action for each event. This is straight forward method for state chart implementation [5]. The entire system will be represented in a class called context class and the event methods are its members. Even though it’s a simple method of state chart implementation, it can't support concurrent states in a state chart diagram. In addition to that, the composite states cannot be implemented using this method, since the state hierarchies cannot be represented in switch case statements.

A different implementation method is proposed by Jakimi [1]. In his approach, the state machine is represented as a class and the states are the attributes of the class. The events in the system are represented as the member functions of the class. An example of this approach is given in figure 2.5. It is a state diagram of an engine which has two states; idle and running. One event in the system is switchON which causes the state transition. The state diagram is implemented as class Engine. An integer attribute, on, is defined to represent the system state. When the system in idle state, on=0; and when the system is in running state, on=1. The event is represented as the member function switchOn() which changes the value of state variable.

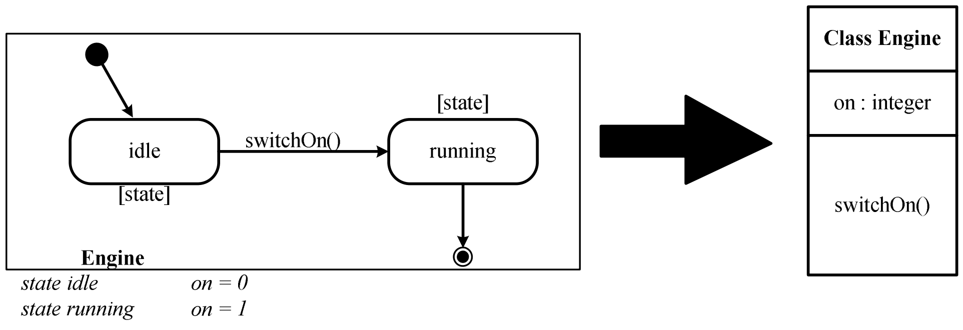


Figure 2.5: Class generated for the state machine

In order to implement composite states and parallel states the language specific features like enumerators have been used in research works. Ali [54] presents the implementation of concurrent and hierarchical state machines by making use of enumerators in Java language. In Java, enums can have data members and member functions similar to class concept. The enum values can override the member functions. Events and states are represented using enumerator variables. Each event and state becomes an enum value. The transitions from states can be implemented as member functions. In [1], the state of a system is implanted as scalar variables and events are represented as methods. Aabidi [79] proposed a method to implement hierarchical-concurrent and history states by combining the methods proposed in [1]and [54]. They aimed to provide a better way to implement state machines leveraging the positive points of both approaches, state machine encapsulated within a single class, code well structured, clear, compact and easy to understand for the first approach and a better identification of the state for the second approach.

2.5.2 State machine implementation using state tables:

Another method for representing state machine is state tables. It is a two dimensional structure like a matrix. Each row represents different states of the system and the columns show the possible events that can happen in the system. Each element in the table shows which action has to be taken when an event occurs and the next state of the system. The Table 2.1 gives the state table of the state chart shown in Figure 2.4. In state 1, if event 2 occurs the state will be changed to state 2.

Table 2.1 : State table structure for UML state chart diagrams

State \ Events	**Event 1**	**Event 2**	**Event 3**	**Event 4**	**Event 5**
State 1	State 1	State 2			
State 2			State 1	State 3	
State 3					State 1

This approach is more convenient for coding simple state chart diagrams and better than switch case method. As the number of states and events increases the table size increases drastically. It is the main drawback of this approach. Moreover the table size does not depend on the number of transitions. Hence the table can be large even though the numbers of transitions are less. This in turn results in wastage of memory.

2.5.3 State machine implementation using state design patterns:

In state design pattern approach, there will be a class diagram pattern that has to be followed for implementing all state chart diagrams [55, 2 , 56]. There will be one class in the pattern which represents the context (domain) of the state chart diagram. The states in the state chart diagram are abstracted in a single abstract class which will act as an interface to the states in the state chart. The events will be the virtual member functions of the abstract state class. Each individual state in the state chart will be represented as the object of the derived class of abstract state class. If there are 'm' states in the state chart, then there will be 'm' different concrete state classes derived from the abstract state class. A sample state design pattern is shown in Figure 2.6.

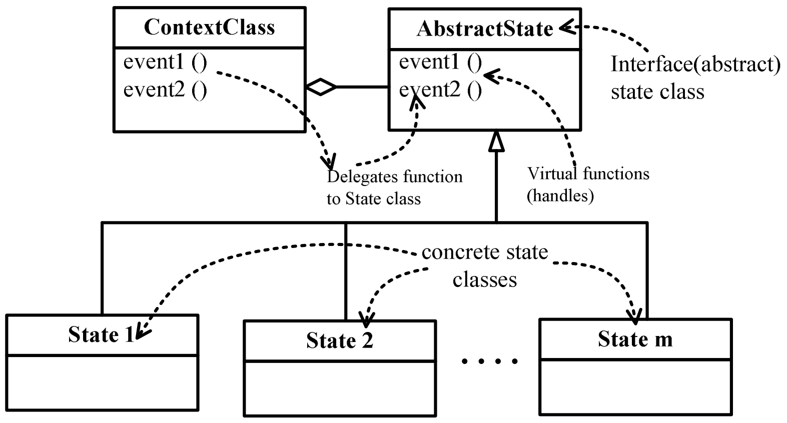


Figure 2.6: Sample State Design Pattern

The object of the context class represents the domain object that needs to be represented in the program. The context class will have a data member (state variable) which represents the current state of the domain object. All the events are represented as member functions of the context class which in turn delegates the function to the corresponding state class objects.

Using state design patterns we can bring the object orientation in the state machine implementation. The domain object, whose state chart is drawn, is implemented as the object of the context class, each state of the domain object is implemented as the object of the corresponding concrete state class. Events are represented as the handles of the abstract state class and the transitions are accomplished by updating the state variables. This approach supports code reusability and avoids redundancy in coding.

There can be variable type of patterns that can be used to represent the state chart diagram. In both the patterns, there is an abstract class which acts as an interface for the state classes. The interface will be connected to the context class. The pattern has an additional object called collaboration object to accomplish the sub states. It is an abstract class which acts as an interface for the sub states [94].

This object oriented approach creates some inconvenience too. In order to add a new state, we have to derive one more concrete state class from the abstract state class. Similarly, to add a new event, we need to add one more virtual function to the abstract state class.

The above mentioned methods failed to represent the concurrent states. Some literature, based on State Design Patterns, attempted to address this issue but resulted in very complex design patterns [131, 54, 133, 132] and failed to implement the key features of state machine. [131] addresses concurrent and hierarchical state implementation. They proposed a double dispatch based event handling. The reaction of the state machine depends on the current state of the system as well as the event occurred. This is the theory behind double dispatch. The implementation pattern presented in this work is very bulky since it requires 17 classes in the implementation for representing a state machine with 6 states and 6 events. It makes the implementation very bulky.

A separate class is added to represent the current state of the system in Niaz [94, 49, 2]. In this proposed approach single event can trigger multiple transitions. This is against the semantics of the UML state machine. UML specifies that one event should be consumed for only one transition.

Some researchers have presented customized methods for state machine implementation like [89, 78]. They defined patterns like HSM (hierarchical state machine) and QHSM (Quantum Hierarchical State Machine) in which states machines are defined as a composition of states, not as inheritance of state class as we see so far. [78] used Quantum Programming paradigm.

Some methods, like [111], extend the HSM pattern method. It presents a template based approach to directly convert the state chart diagrams to the C++ code. This method avoids the use of separate code generation tools for the state machine to code conversion. The generated code is optimized using in-lining. The advanced features like concurrent states and history states do not supported by [111].

Combining state chart diagrams with other behavioral diagrams is a different approach to improve the code generation from the system models. [56] is an attempt to make use of state chart diagrams and activity diagrams together for the code generation. Ali [56] proposed a method to implement the dynamic behavior of an application. State transition diagrams and activity diagrams are used for modeling the dynamic behavior. The state of the system is represented as object and the state transition is implemented as method. Similar to the previous methods, the state hierarchy and concurrency are implemented using inheritance and composition.

Direct execution of the state chart diagrams are also investigated by few researchers [126, 57] etc. Schattkowsky [126] demonstrates how a fully featured UML 2.0 state machine can be represented using a small subset of the UML state machine features that enables efficient execution. They are trying to directly execute the state machines without converting it to implementation code. It is an alternative to native code generation approaches since it significantly increases portability. [57] presented a method to generate C++ code from the State chart. State chart is modeled in XML and then using Python the XML document is parsed to Python object. In the next step a templating engine is used to convert the parsed XML to C++ code. For this conversion they use a pattern based approach. The pattern contains state controller, state chart, state and events.

2.6 Summary

In this chapter, a review of literature on code generation techniques from different UML models is carried out. The review was done for static models as well as dynamic models. The structural models can contribute to the structure of the software system and the dynamic models are most important to generate the source code for the behaviour of the software system, mainly the control flow and method definitions. The code generation approaches for use case diagram, sequence diagram, activity diagram and state diagram, have been reviewed in detail. Based on the review of code generation based works, some inferences were drawn.

The main issues observed during the study of use case and sequence diagram based code generation were:

- ❖ Use case diagrams are not considered for modularizing the generated code.
- ❖ The relationships between use cases, like <<extend>> and <<include>>, is not taken into account for code generation.

- ❖ An algorithm based code generation from use case and sequence models is not adopted in the literatures.
- ❖ Combining sequence models with other model for the better expressiveness of the software system design is not addressed in the literatures.

Activity models are the best tool to represent the process flow in a system. The control flow of the source code can be generated from this model. The main issues observed during the study of activity models based code generation were:

- ❖ Enhancing activity models with additional information for better code generation is not addressed well in the literature.
- ❖ Combining activity models with other model for the better expressiveness of the software system design is not addressed in the literatures.
- ❖ An algorithm based code generation from activity models is not adopted in the literatures.

UML state chart diagram is a very strong tool to model embedded systems. From these state chart diagrams, we can generate complete code for the system. Some advanced features, like history state, are not so far considered for code generation, or an effective method is not proposed so far. The main issues observed during the study of state chart models based code generation were:

- ❖ Existing design patterns for state machine implementation are complex and customized for a problem and so difficult to reuse in other scenarios.

- The available design patterns do not support the concurrent states and the history states.

It is evident from the literature review that, there are more opportunities to improve code generation from the UML behavioural models by solving the above issues.

Chapter 3 UML BEHAVIOURAL MODELS

3.1 Introduction

UML forms a de-facto standard for software system design [99]. It is used for high level system design. It is a modeling language based on Model-driven engineering [33, 115, 123] concepts and consists on the application of models to raise the level of abstraction in which developers create software with the objective of making easier to cope development processes with the required standardized methodologies. The application of these techniques improves software quality, reduces the problem and creates a possible solution in the developer perspective. This higher level of abstraction offered by models also leads to a better reuse of software business logic. In addition to these advantages, it is possible to use tools to transform UML models into other models (meta-models) as well as source code. This will lead to a reduction of software development time, costs and preventing diagram's data losses due to misinterpretation of the model during code generation.

UML can be used to model the architecture and behavior of any kind of software project. This is due to the fact that UML provides many different diagrams or views of a system: class, component and deployment diagrams focus on different aspects of the structure of a system while the behavioral diagrams such as use case, state chart, activity and interaction diagrams focus on its dynamics. All the behavioral diagrams except use case diagrams are closely related. We can convert a collaboration diagram into a sequence diagram and vice versa. State charts are used as the semantic foundation of the activity diagrams and it is possible to represent an execution (a trace) of a state chart or an activity diagram as a sequence or collaboration diagram [34].

UML provides structural and behavioural diagrams to design a system [34, 58]. Structure diagrams show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. These static structures are represented in UML by class diagrams, object diagrams, component diagrams and deployment diagrams.

The behaviour diagrams show the dynamic behaviour of the objects in a system, which can be described as a series of changes to the system over time. UML provides five types of behavioural diagrams; use case diagram, sequence diagram, collaboration diagram, activity diagram and state chart diagram [34].

3.2 Behaviour Models

The internal behaviour of a system is described using the behavioural models. The UML sequence diagrams and activity diagrams are used for business process modeling. The change in the data (state) is modeled using the state chart diagrams.

The system modeling starts with the identification of different use cases in the problem. Next, we have to identify the objects and their interactions to achieve the use cases. The sequence of messages that pass between the objects is described in the sequence diagram. It helps the users as well as programmers in understanding real-time specifications and complicated use cases. The state of the objects may change in response to an event. These changes in the state of the objects are represented using state chart diagrams.

3.2.1 Use case diagram

Use case diagrams are used to represent the use case or functionalities of a system. It represents the high level requirements that the system fulfills. It has three main components; use cases, actors and relationships.

Use cases represent the functional requirements of the system. Actors are the controllers who interact with the use cases. Relationships (or associations) exist between actors and use cases as well.

A sample use case diagram which shows the main components of the use case diagram is given in figure 3.1.

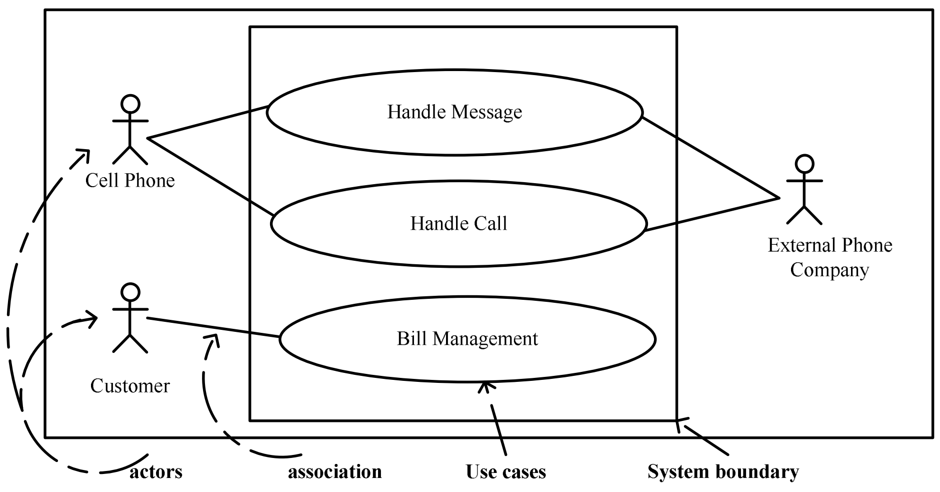


Figure 3.1: Sample Use Case diagram

Use cases help in organizing and defining the software. So the code generation can be started from this model.

3.2.2 Sequence diagram

It is the most commonly used interaction diagram. It basically represents the interactions or sequence of messages between objects to accomplish a specific functionality or use case of the system. It has five main components; actors, objects, lifelines, messages and guards.

A sample sequence diagram which shows the main components of the sequence diagram is given in figure 3.2.

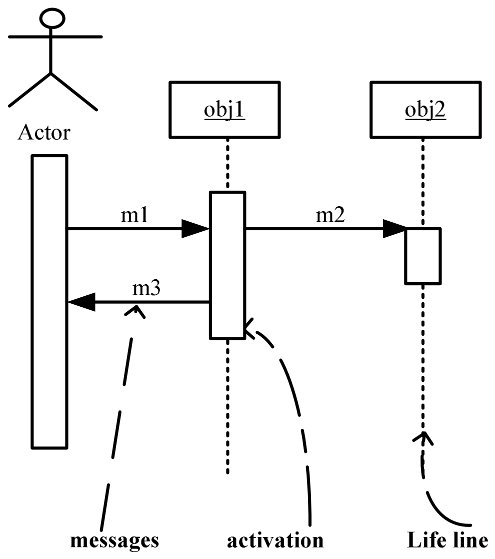


Figure 3.2: Sample sequence diagram

The sequence diagram visualizes the logic behind sophisticated functions [109]. They are used to describe how a use case is achieved through object (or component) interactions. So linking sequence diagrams with the use cases will help to add more execution logic in the implementation code. Sequence diagrams are best suited for implementing the control classes since it represents the execution logic of each use case and the objects involved in it.

Collaboration diagram is similar to sequence diagram. It basically shows the object interactions and the organization of the objects as well. We can convert sequence diagram to collaboration diagrams and vice versa without losing any information.

3.2.3 Activity diagram

The control flow of the entire system is described using activity diagrams. It gives a clear picture of how the system will work when executed. That is, the activity diagram describes the control flow between different activities. Activities are the functions of the system and the control flow can be sequential, concurrent or branched.

The components of the activity diagrams are initial and final nodes, activity node, control flow, decision node, fork and join nodes, merge node, swimlanes and time event node as shown in figures 3.3 to 3.12.

Activity Diagram Notations

Initial Node – The starting state before an activity takes place is depicted using the initial node (Figure 3.3). A process can have only one initial state unless we are depicting nested activities. We use a black filled circle to depict the initial state of a system. For objects, this is the state when they are instantiated. The initial node from the UML Activity Diagram marks the entry point and the initial Activity State.

Figure 3.3: Notation for initial node

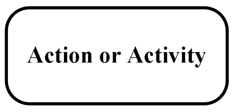


Figure 3.4: Notation for an activity state

Action or Activity Node – An activity represents execution of an action on objects or by objects. We represent an activity using a rectangle with rounded corners (Figure 3.4). Basically any action that takes place is represented using an activity.

Action Flow or Control flows – Action flows or Control flows are also referred to as paths and edges. They are used to show the transition from one activity state to another (Figure 3.5). An activity state can have multiple incoming and outgoing action flows. We use a line with an arrow head to depict a Control Flow. If there is a constraint to be adhered to while making the transition it is mentioned on the arrow.

Decision node and Branching – When we need to make a decision before deciding the flow of control, we use the decision node (Figure 3.6). The outgoing arrows from the decision node can be labeled with conditions or guard expressions. It always includes two or more output arrows.

Figure 3.5: Notation for control Flow

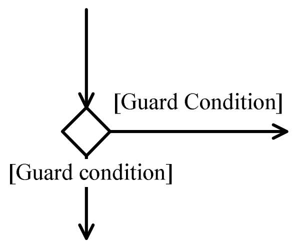


Figure 3.6: Guards being used next to a decision node

Guards – A Guard refers to a statement written next to a decision node on an arrow sometimes within square brackets (Figure 3.6). The statement must be true for the control to shift along a particular direction. Guards help us know the constraints and conditions which determine the flow of a process.

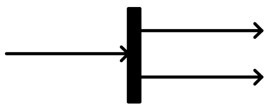


Figure 3.7: Notation for fork

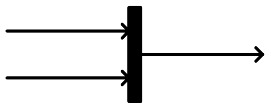


Figure 3.8: Join notation

Fork – Fork nodes are used to support concurrent activities (Figure 3.7). When we use a fork node when both the activities get executed concurrently i.e. no decision is made before splitting the activity into two parts. Both parts need to be executed in case of a fork statement. We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent activity node and outgoing arrows towards the newly created activities.

Join – Join nodes are used to support concurrent activities converging into one. For join notations we have two or more incoming edges and one outgoing edge (Figure 3.8).

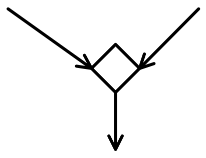


Figure 3.9: Notation for merge node

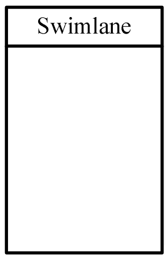


Figure 3.10: Swimlanes notation

Merge or Merge Event – Scenarios arise when activities which are not being executed concurrently have to be merged. We use the merge notation for such scenarios. We can merge two or more activities into one if the control proceeds onto the next activity irrespective of the path chosen (Figure 3.9).

Swimlanes – We use swimlanes for grouping related activities in one column. Swimlanes group related activities into one column or one row. Swimlanes can be vertical and horizontal. Swimlanes are used to add modularity to the activity diagram. It is not mandatory to use swimlanes. They usually give more clarity to the activity diagram. It's similar to creating a function in a program. It's not mandatory to do so, but, it is a recommended practice. We use a rectangular column to represent a swimlane as shown in the Figure 3.10.

Time Event – We can have a scenario where an event takes some time to complete. We use an hourglass to represent a time event (Figure 3.11).

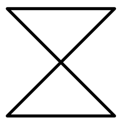


Figure 3.11: Time event notation

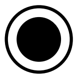


Figure 3.12: Notation for final Node

Final Node or End Node – The state which the system reaches when a particular process or activity ends is known as a Final node or End node (figure 3.12). We use a filled circle within a circle notation to represent the final state in an activity diagram. A system or a process can have multiple final nodes.

A sample activity diagram which shows the main components of the activity diagram is given in figure 3.13. This figure shows the initial and final nodes, the decision node, merge node, fork node, join node, activity nodes and the control flows.

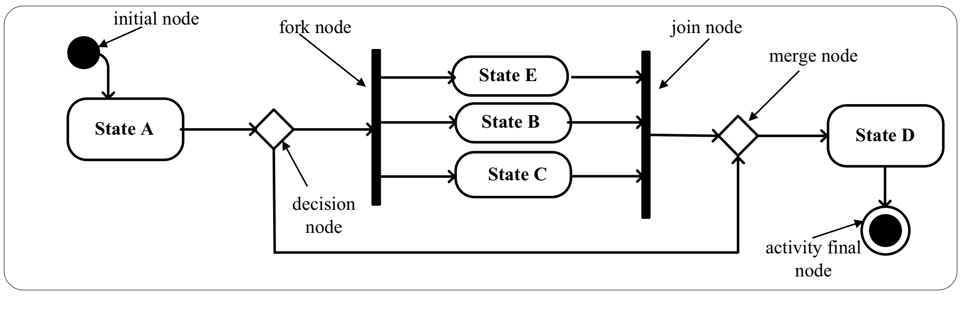


Figure 3.13 : Sample Activity diagram

Activity Diagrams are essential in code generation since they describe control flow of how use cases are achieved, by depicting conditions, constraints, sequential and concurrent activities. Apart from sequence diagram, activity diagrams can add more information to implementation code like, the concurrent activities, conditions and sequence of activities inside and among the objects. Activity diagrams show the various steps involved in UML use cases. It can also give the constraints, conditions and logic behind algorithms.

The combination of use case diagram and the sequence diagram is used to prepare the prototype of the system. The details in the activity diagram will help us to incrementally add method definitions to the prototype to evolve it as the final software.

3.2.4. State chart diagram

The state chart diagram represents the event driven state changes of a system components (or objects). It visualizes the reaction of a system by internal/external factors.

State chart diagrams represent the events responsible for state changes and the different states of the objects (or components) in the system.

The basic components of the state diagram are initial & final nodes, state node, fork and join nodes and transitions labeled with events.

Initial state – We use a black filled circle represent the initial state of a System or a class (Figure 3.14).

Figure 3.14: Initial State

Figure 3.15: Transition

Transition – We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labeled with the event which causes the change in state (Figure 3.15).

State – We use a rounded rectangle to represent a state. A state represents the conditions or circumstances of an object of a class at an instant of time (Figure 3.16).

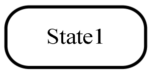


Figure 3.16: State notation

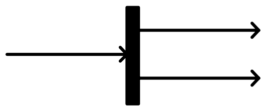


Figure 3.17: A diagram using the fork notation

Fork – We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states (Figure 3.17).

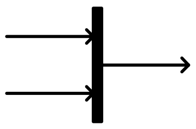


Figure 3.18: Join notation

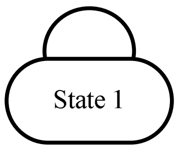


Figure 3.19 : Self transition notation

Join – We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events (Figure 3.18).

Self transition – We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases (Figure 3.19).

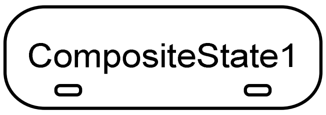


Figure 3.20: A state with internal activities

Figure 3.21: Final state notation

Composite state – We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state (figure 3.20).

Final state – We use a filled circle within a circle notation to represent the final state in a state machine diagram (figure 3.21).

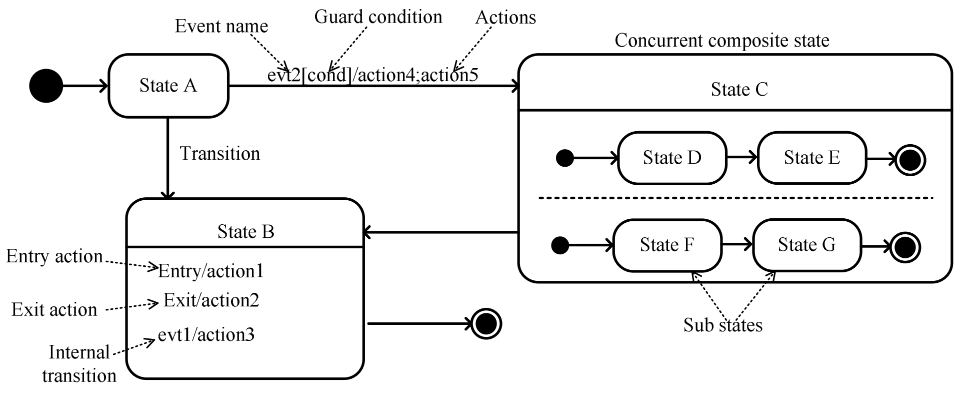


Figure 3.22: A sample State chart diagram

A sample state chart diagram which shows the main components of the state chart diagram is given in figure 3.22. It has three states, States A, B and C. States A and B are simple states. State C is a composite state with two parallel regions. Each transition in the state transition diagram is labeled with the event name which causes the state transition, the guard condition (optional) and the actions to be executed. Inside each state we can specify the entry action, exit action and the internal actions.

UML State chart diagrams represent the state changes of the objects or entities in the system. So, it is best suited to generate code for entity classes. Entities are the objects representing the system data. Moreover, state chart diagrams can be used to generate code for embedded system where we can represent the states of the whole system in a single state chart diagram.

3.3 Use of Object Constraint Language (OCL)

UML is not a fully formal language. Its semantics are not fully formalized. In many places natural language is used for model specification. It leads us to a scenario where the precise model presentation is difficult. So, whenever we use activity diagram, or any UML diagram, for code generation, it is recommended to complement it with specification languages like Object Constraint Language (OCL) [98, 7, 138]. OCL can supplement some of the shortcomings of UML notations, like lack of precision. We need to give the constraints of the objects in the UML models. Usually, these constraints are given in natural languages. This may be ambiguous. A formal language is required to unambiguously present the constraints. Since OCL is a formal language and all constructs in OCL are well defined, it can unambiguously specify the constraints on the object or system. At the same time OCL is familiar and widely used in software industry.

OCL is a formal language used to express constraints. These typically specify invariant conditions that must hold for the system being modeled. Note that when the OCL expressions are evaluated, they do not have side effects.

A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a string mathematical back ground, but difficult for the average business or system modeler to use.

OCL has been developed to fill this gap. It is a formal language that remains easy to read and write. It has been developed as a business modeling language within the IBM Insurance division and has its roots in the Syntropy method.

OCL is a pure expression language; therefore, an OCL expression is guaranteed to b e without side effect. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model. This means that the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to specify a state change (e.g., in a post-condition).

OCL is not a programming language; therefore, it is not possible to write program logic or flow control in OCL. You cannot invoke processes or activate non-query operations within OCL. Because OCL is a modeling language in the first place, not everything in it is promised to be directly executable.

OCL is a typed language, so that each OCL expression has a type. To be well formed, an OCL expression must conform to the type conformance rules of the language. For example, you cannot compare an Integer with a String. Each Classifier defined within a UML model represents a distinct OCL type. In addition, OCL includes a set o f supplementary predefined types.

As a specification language, all implementation issues are out of scope and cannot be expressed in OCL. The evaluation of an OCL expression is instantaneous. This means that the states of objects in a model cannot change during evaluation.

OCL can be used for a number of different purposes: To specify invariants on classes and types in the class model; To specify type invariant for Stereotypes; To describe pre- and post conditions on Operations and Methods; To describe Guards; As a navigation language; To specify constraints on operations

The use of OCL improves the clarity of specification. In this regard, researchers show interest in usage of OCL. OCL is a pure expression language which uses expressions similar to object oriented languages [81, 125]. The evaluation of the OCL expression [16, 17, 47] will not change the state of the system. So, it is safe to use the OCL expressions in the UML diagrams and they are said to be side-effect free. The OCL statements are not directly executable since it is a modeling language. Moreover, all OCL statements should conform to the type conformance rule of the language.

3.4 Summary

This chapter presented the UML behavioural models which are used to model (design) the behaviour of a system. The core UML behavioural diagrams are use case diagram, sequence diagram, collaboration diagram, activity diagram and state chart diagram.

The overall functions and actors of the software is given in use case diagrams. So the code generation can be started from this model. It helps us to realize the boundary and control classes.

Sequence diagrams are best suited for implementing the control classes since it represents the execution logic of each use case and the objects involved in it. They are used to describe how a use case is achieved through object (or component) interactions. So linking sequence diagrams with the use cases will help to add more execution logic in the implementation code.

Activity Diagrams are essential in code generation since they describe control flow of how use cases are achieved, by depicting conditions, constraints, sequential and concurrent activities. Apart from sequence diagram, activity diagrams can add more information to implementation code like, the concurrent activities, conditions and sequence of activities inside and among the objects. Activity diagrams show the various steps involved in UML use cases. It can also give the constraints, conditions and logic behind algorithms.

UML State chart diagrams represent the state changes of the objects or entities in the system. So, it is best suited to generate code for entity classes. Entities are the objects representing the system data. Moreover, state chart diagrams can be used to generate code for embedded system where we can represent the states of the whole system in a single state chart diagram.

So, the UML behavioural diagrams named use case diagram, sequence diagram, collaboration diagram, activity diagram and state chart diagram together with class diagram can generate implementation code for the boundary, control and entity classes.

Chapter 4 CODE GENERATION FROM UML USE CASE MODELS

4.1 Introduction

In software development life cycle, requirement specification is a critical phase. The correctness, completeness and consistency of the requirement specifications need to be checked. Otherwise it may lead to the failure of the project itself. Prototyping is a widely accepted approach for gathering customer requirements completely and for testing it [141, 100, 66, 24]. The prototype of the system will be made after collecting the customer requirements. These initial requirements will be refined after testing the prototype. The prototype will be modified based on the customer reviews on the requirements. The updated version of the prototype will again be tested by the customer and this process repeats till the customer gets satisfied [108]. This process is shown in Figure 4.1.

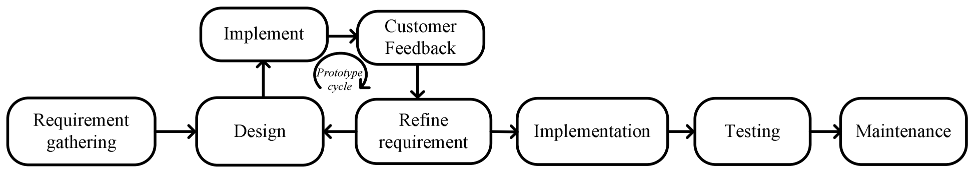


Figure 4.1: Software Life Cycle

The use case model represents the business use cases of the system and gives the important functionalities and their relationships. UML use-case models are used primarily for the requirement analysis in software development [38]. It draws the external view of a software system. It describes the functionalities of the system which are used or initiated by a human user.

A use-case description is associated with each use-case in the use-case diagram which explains the name of the use-case, the summary of its working, the actor (human user) who uses the use-case, the pre and post conditions, the description of the use-case and the alternative options in the use-case. All these details will be given in natural languages like English [116].

The use case model is used throughout the software development. In the requirement specification phase, it is used for specifying functional requirements. This will be used in analysis and design phase as the base input. Moreover, the use case model is used as input to iteration planning, for test case generation and as a major component for user documentation.

The readers may wonder how a description with no technical details can contribute to prototype generation. There are a few research works going on, to directly convert the software requirements to source code. Our focus is on Model Driven Development [113, 123, 3, 73, 88, 115, 117] and Executable UML [87] since these two terms are welcomed in the software industry and are widely accepted these days. Our aim is to convert the use-case model to prototype of the system. In this view, we use use-case diagrams along with sequence diagrams for prototype generation. Most of the research works in this area concentrate only on sequence diagrams and class diagram for code generation [83, 102, 107]. The use case diagrams are very useful and informative for the clients. Since the functional requirements are marked in the use case diagrams, any change in requirement will also be reflected in it. So the initial stage of code generation, especially the prototype generation, should take use case diagram as one of its inputs. This will help us to modularize the generated code based on the functional requirements and so it gives traceability from design to code.

The use cases are summarized to a software package which satisfies the required functionalities of the system. It helps to map the system requirements with the features provided by the system. Sequence diagram alone cannot project the customer requirements satisfied by the implemented system. That's why we included use case diagram for code generation.

The main contribution of this chapter:

- Presents a method to generate code from UML use case diagram, where use case scenarios are expanded with sequence diagrams.

4.2 Requirements Modeling With Use Cases

A use case diagram is the graphical representation of the functional requirements of a proposed system [59]. It gives the scope of the proposed system. It is mainly used for communicating with the end-users of the system. So it is always kept non technical.

The two components in a use case diagram are the use cases and the actors. A use case is a single unit of meaningful work. That means, one aspect of the behavior of a system is represented by a use case. A use case can be described with diagrams or textual descriptions. A description includes the requirements, constraints, scenarios and scenario diagrams. UML sequence diagram is used as the scenario diagram. The users of the system are marked as actors in use case diagram. An actor in a use case diagram can be something with a behavior or role. For example, a person, another system, organization etc. Actors interact with the system. The use case diagram describes how actors related to use cases. The use case diagram helps to package the user specific methods in the actor classes during code generation.

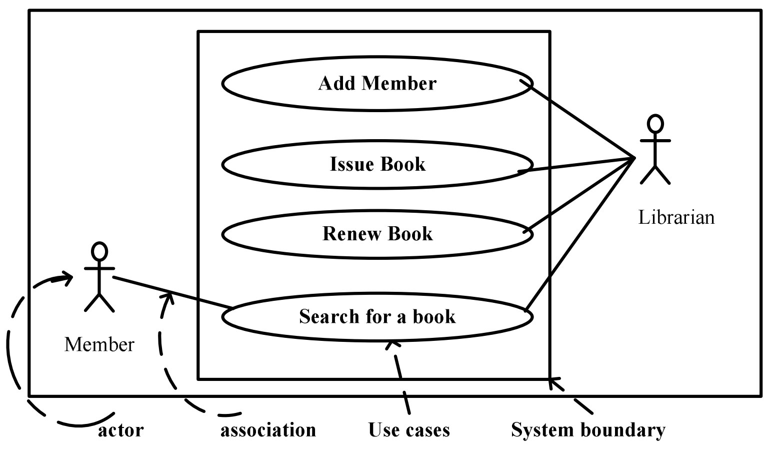


Figure 4.2: Sample Use Case diagram

A sample use case diagram along with its different components is shown in Figure 4.2. The use cases are *Add Member*, *Issue Book*, *Renew Book and Search for a Book*. The actors are the *Member* and *Librarian*. The lines between the use cases and the actors are called association. The proposed system is represented as the rectangular box in which the uses cases are embedded.

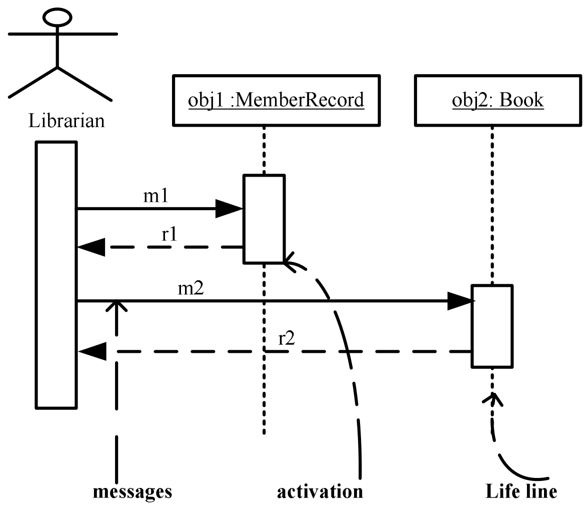


Figure. 4.3: Sample sequence diagram

Sequence diagram represents the interaction between different objects and actors in a system to accomplish a functional requirement of the system

[59]. The basic components in the sequence diagram are objects, actors, lifeline & activation and messages. Objects and actors are the instances of classes. The existence of an object is represented by lifeline (the dashed vertical lines). The rectangular box on the life line shows the activation period of the object. Messages are represented with arrows from the life line of one object to another. It indicates the communication between objects.

A sample sequence diagram is shown in the Figure 4.3. Three objects are there, one actor and two other objects; obj1 and obj2. Usually, in each sequence diagram the communication starts from an actor. Here, m1 and m2 are the messages. The reply of the messages are marked using dotted arrows. Here, r1 and r2 are the replies. Normally there will be a sequence number along with each message to show their order. The dashed line from each object (vertical) is called the life line. And the rectangular boxes on the life line shows the activation period.

4.3 Code Generation

The static structure of a system is presented using UML use case diagram and class diagram. The dynamic and behavioral aspects of the system are presented using UML sequence diagrams. Sequence diagram emphasizes the time ordering of message between objects.

For code generation, we first analyze the system requirements and prepare use case diagram with necessary scenario descriptions. Then develop the class diagram to show the system architecture. Further, the behavioral details of the system are designed using sequence diagram. Finally, using the information in all the three diagrams generates code out of it [109, 46].

4.3.1 Code generation from use case

The use case diagram is used in code generation to identify different services provided by the proposed system. Each service is mentioned as a use case in the diagram. Each use case will be associated with one or more actors in the system. The use cases are converted to service methods in the associated actor classes. The steps, shown in figure 4.4, are used to generate the code from the use case diagrams.

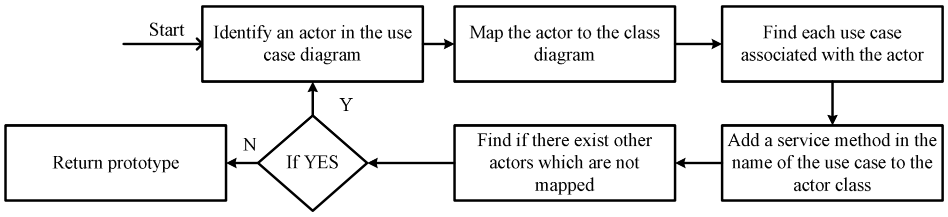

Start
Identify an actor in the use case diagram
Map the actor to the class diagram
Find each use case associated with the actor
Add a service method in the name of the use case to the actor class
Find if there exist other actors which are not mapped
If YES
Y
N
Return prototype

Figure 4.4 : Steps for code generation from UML use case diagram

As the first step identify an actor in the use case diagram. The name of actor is directly available in the <actor> tag of the XML representation of the use case diagram. Then check whether the class diagram contains the class of the actor. If not create the class. The actor may associate with multiple use cases. Each use case is added to the actor class as a service method. The relationships like ‘extend’ and ‘include’ with other use cases are also mapped to the Actor class. Since the ‘extend’ relation is optional, the extended use case will be called only when the extension condition is satisfied. So the corresponding sub service call will be given inside *if statement*. The relation ‘include’ is mandatory, so the ‘*included’* sub service call in made inside the service method. After adding all use cases search for another actor in the use case diagram which is not mapped to the class diagram. Repeat this procedure till we map all actors to the class diagram.

In short, each use case is converted to a method definition named as service method. Each service method is expanded using the sequence diagram which depicts the scenario.

4.3.2 Code generation from sequence diagrams

A sequence diagram can be formally defined as follows [76, 121].

$$
\begin{aligned}
SD &= (\,\mathcal{o},\ m\,)\ where\\
&\quad \mathcal{o} = \{x \mid x \text{ is an object / actor}\}\\
&\quad m = \{msg \mid msg \text{ is a message}\}\\
&\quad\ \ \text{where } msg \text{ is a tuple.}\\
&\quad msg = \left(\,ob_i : C_i,\ ob_j : C_j,\ action,\ order\right)
\end{aligned}
$$

The sequence diagram is a tuple which includes set of objects (o) and set of messages (m) send between the objects. A message, *msg*, contains the sender object ob_i of the class C_i, the receiver object ob_j of the class C_j, the method call *action* and finally the sequence *order* of the message. These details are essential for code generation from sequence diagrams. The same notations have been used in the code generation algorithms proposed in this chapter.

The sequence diagram is converted to the hierarchical structure tree [18, 104] before code generation. Structure tree is nothing but a tree structure of the messages passed between the objects. The nodes in the tree are the objects which are participated in the communication and the edges are actually the messages passed between the objects. The root of the tree will be one of the actors in the use case diagram. A sample sequence diagram and its hierarchical structure tree (HST) are shown in Figures 4.5 and 4.6.

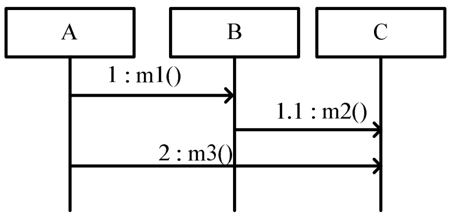


Figure 4.5: Sample sequence diagram

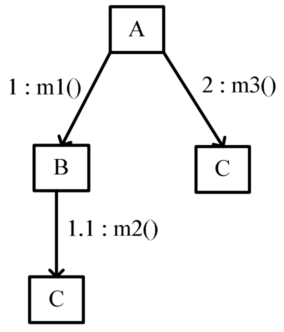


Figure 4.6: Hierarchical structure tree

The code generation from sequence diagram has two passes. In the first pass, identify the messages and populate the classes by adding the corresponding method in the receiver class. In the second pass, the service methods will be defined.

The rest of this section presents three algorithms to generate code from the sequence diagrams. Algorithm 4.1 generates the service functions corresponds to each use case or sequence diagram that represents the scenario of a use case. Algorithm 4.2 is used to populate each class with the method declarations based on the messages passed between objects as depicted in the sequence diagram. Algorithm 4.3 generates the method definitions.

Algorithm 4.1 Generate Services

***Algorithm 4.1 generate_services** (HST,P)*

```
    Input    :    HST,P
    Output   :    P
 1:  Parse hierarchical structure tree, HST
 2:  Let serviceName=HST.scenarioName
 3:  Add the statement "serviceName(){ System.out.println (""+serviceName);
                                        //TODO code here"   to P
 4:  Get the root node of HST.
 5:  Let count = 1.
 6:  Take the message, msg, with msg.order==count.
 7:  receiverObj=msg.obj_j, fnName=msg.action.
 8:  Update P by populating the method serviceName() with "receiverObj.fnName();"
 9:  Repeat steps 6 to 8 till there are no more messages to read from the root.
10:  Close the method definition
11:  Return P
```

The algorithm *generate_services* ( ) takes *HST* and the prototype *P* as input and returns the updated *P*. Each *HST* represents a scenario of the use case. This scenario name will be taken as the service name. Service is the method which implements a scenario of a use case. The service methods will be added to the context class. Each service method is defined as a series of method calls from the actor object. The sequence of the method calls are decided by the order of the message, *msg.order*. The method calls are added as *objName.fnName*(). Here *objName* we get from $msg.obj_j$ and *fnName* from *msg.action*. Repeat this for all messages going from the root. Then return the updated prototype P.

Algorithm 4.2 Populate classes

The algorithm *populate_classes*( ) is used to add the method declaration to the classes based on the messages passed between objects of each class. The algorithm takes hierarchical structure tree (*HST*) of the sequence diagram and the class diagram *CD* as input. The output of the algorithm is the modified class diagram.

Algorithm 4.2 *populate_classes(HST,CD)*

```
     Input    :    HST,CD
     Output   :    CD

 1:  Parse hierarchical structure tree, HST
 2:  Go to the root node, say cur_node.
 3:  Read message from the edge of cur_node, say msg.
 4:  Search class msg.C_j in CD for a method named msg.action.
 5:  if it does not exists add "msg.action() { System.out.println (""+msg.action);
                               // TODO code here }" to class msg.C_j
 6:  Enqueue ob_j to queue, Q_pc.
 7:  Read next message edge of cur_node.
 8:  Repeat steps 4 to 7 until there is no more edge to traverse.
 9:  Dequeue Q_pc to cur_node,
10:  Repeat steps 3 to 9 until Q_pc is empty.
11:  return CD
```

First of all the *HST* is parsed to retrieve the data in the tree. Go to the root node which is normally an actor in the use case diagram. Take it as the current node, *cur_node*. From this point we start a breadth first traversal in the HST. Take each message edge of the *cur_node* and find the *msg.action*. This represents a method call. Check whether this method is already declared or defined in the receiver class, *msg.*C_j. If not, add a method declaration for *msg.action* in that class. Similarly take each node and find all message edges starting from it and checks whether those have corresponding method declarations in the respective receiver classes. Q_{pc} is the queue used for executing the tree traversal. After checking each message the receiver object ob_j is enqueued in Q_{pc}. Each time, an object is dequeued from Q_{pc} and find out the messages associated with it. Enqueue all receiver objects of those messages to Q_{pc}. Continue this till the queue is empty. Then return the updated *CD*.

Algorithm 4.3 populate methods

```
Algorithm4.3  populate_methods(HST,CD)

      Input   :     HST,CD
      Output  :     CD

  1:  Parse hierarchical structure tree, HST
  2:  cur_node = root node.
  3:  Take one unread message, msg, from cur_node and mark as read.
  4:  If there exists an unread message, then go to step 5, else go to step 10.
  5:  Open class msg.C_j and method named msg.action
  6:  push cur_node to stack.
  7:  cur_node = msg.C_j.
  8:  find all messages from cur_node.
  9:  If any message exists,
          9.1:  take one message, msg, and fnName=msg.action,
                obj=msg.ob_j
          9.2:  Append a statement "obj.fnName();" to the method opened
                in step 5
          9.3:  Repeat steps 9.1 and 9.2 till there is no more message to
                take.
 10:  else go to step 12
 11:  Repeat steps 3 to 9 till there are no more messages to read.
 12:  cur_node = popStack( ).
 13:  Repeat steps 3 to 12 till the stack is empty.
 14:  Return CD
```

This algorithm takes *HST* and *CD* as input. It returns the class diagram with updated method definitions. The parsed hierarchical structure tree is used for updating method definitions. First the toot node is taken as the current node, *cur_node*. There can be many messages initiating from *cur_node*. These messages will be taken one by one. Already considered messages are labeled as ‘read’. If there exists an unread message, then open the receiver class (*msg.*C_j) of the message and the method definition that is being called (*msg.action*). then push the *cur_node* to stack for future reference. Set new *cur_node* as receiver node (*msg.*C_j). Find all messages initiated from the *cur_node* and append the corresponding method call statements to the method

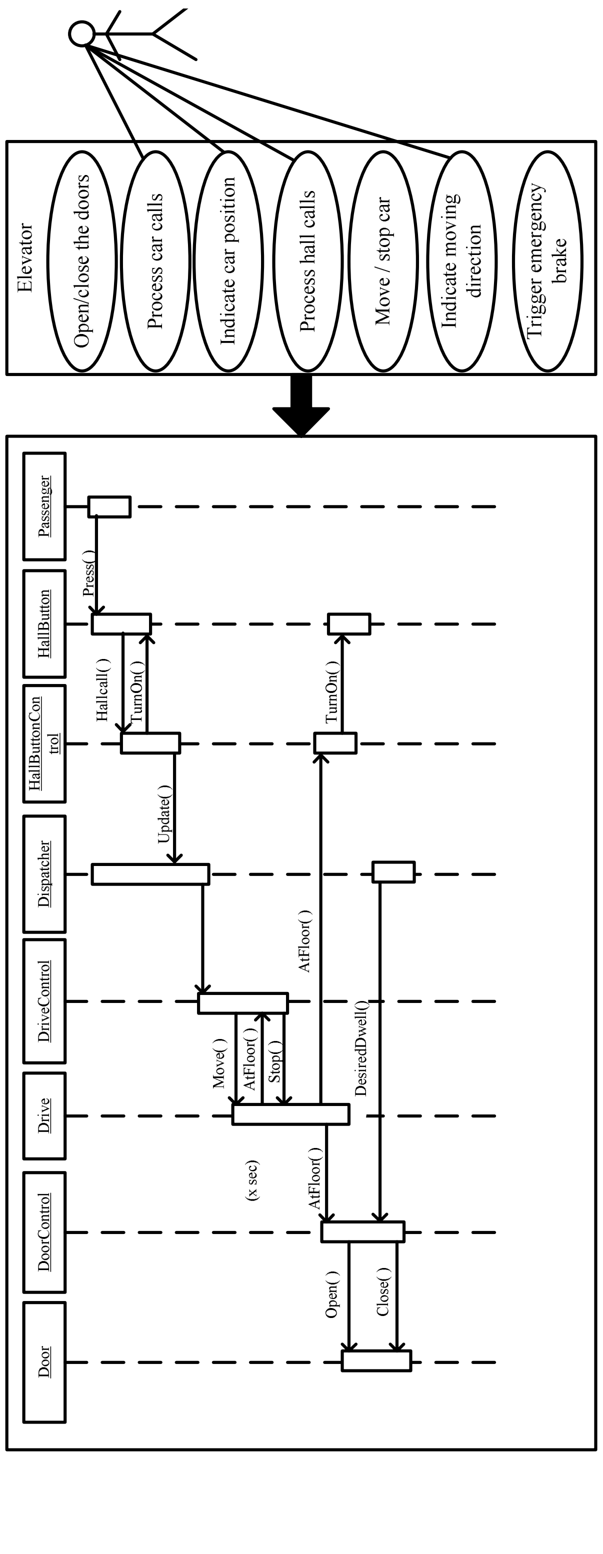


Figure 4.7: Use-Case diagram of Elevator System and the sequence diagram for process hall call operation

definition. Repeat the whole process until there are no more messages to read. Then, backtrack to the parent node by popping it out from the stack and set as *cur_node*. Finally return the updated class diagram.

In the next section we present a case study to demonstrate the working of the above algorithms.

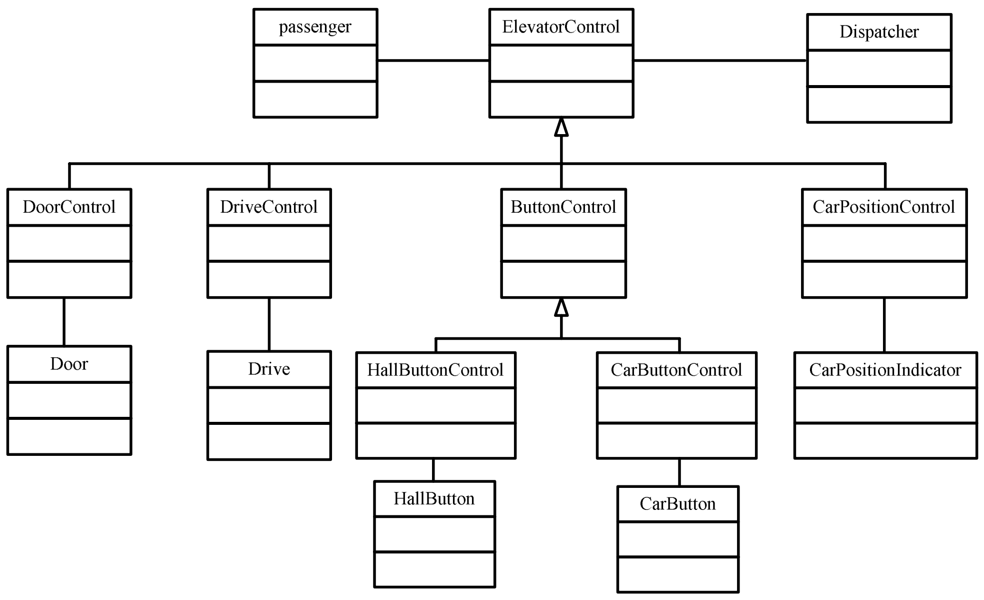


Figure 4.8: Class diagram for elevator system

4.4 Elevator System as Case Study

In this section we present a case study, the elevator system. We consider a few basic features of an elevator; open and close doors, process car calls, indicate car position and process hall call.

The door open and close operation will be done automatically by the door. The passenger need not initiate it. Car calls means, inside the elevator there are buttons corresponding to each floor. The passenger can choose his destination floor using these buttons. It is called car call. The passenger initiates the car call operation. The third operation, indicate car position, is done automatically by the door. The last operation, hall call operation is done by the passenger. In every floor there is two buttons. The passenger can press any one button depending on the direction he wants to go, up or down.

The four features are presented in the use case diagram in Figure 4.7. The sequence diagram for the use case 'process hall calls' is also shown in the figure. The class diagram for the elevator system implementation is shown in Figure 4.8.

The service methods will be generated by the algorithm 4.1. Here in the case study we have shown the service 'process hall calls'. A method in this name will be generated in the context class which will be then delegated to the actor class Passenger. The code generated from Algorithm 4.1 is shown in Figure 4.9.

```
serviceProcessHallCall(){
        System.out.println("serviceProcessHallCall");
        objHallButton.press();
}
```

Figure 4.9: code generated by Algorithm 4.1

From the class diagram the class skeleton can be generated using any CASE tool. Then apply the algorithm 4.2 to populate the classes. The algorithm takes the class diagram and the sequence diagram structure tree as input. The code generated from Algorithm 4.2 is shown in figure 4.10.

We have to call the algorithm for each sequence diagram of the system. For example, here we show the sequence diagram for processing hall call operation. The communication for processing hall call operation includes the objects of the classes like, HallButton, HallButtonControl, Dispatcher, DriveControl, Drive, DoorControl and Door. These classes will be updated by algorithm 4.1.

```
        class HallButton{
                        press() { System.out.println("press");
                                //TODO CODE HERE
                        }
                        turnOn() { System.out.println("turnOn");
                                //TODO CODE HERE
                        }
                        turnOff() { System.out.println("turnOff");
                                //TODO CODE HERE
                        }
                }

                class HallButtonControl{
                        hallCall() { System.out.println("hallCall");
                                //TODO CODE HERE
                        }
                        atFloor() { System.out.println("atFloor");
                                //TODO CODE HERE
                        }
                }
                class DriveControl{
                        desiredFloor() {
System.out.println("desiredFloor");
                                //TODO CODE HERE
                        }
                        atFloor() { System.out.println("atFloor");
                                //TODO CODE HERE
                        }
                }
```

```
class Drive{
        move() { System.out.println("move");
                //TODO CODE HERE
        }
        stop() { System.out.println("stop");
                //TODO CODE HERE
        }
}
class DoorControl{
        atFloor() { System.out.println("atFloor");
                //TODO CODE HERE
        }
        desiredDwell() { System.out.println("desiredWell");
                //TODO CODE HERE
        }
}
class Door{
        open() { System.out.println("open");
                //TODO CODE HERE
        }
        close() { System.out.println("close");
                //TODO CODE HERE
        }
}
class Dispatcher{
        update() { System.out.println("update");
                //TODO CODE HERE
        }
}
```

Figure 4.10: Code generated by Algorithm 4.2

```
class HallButton{
        press(){System.out.println("press");
                //TODO CODE HERE


                objHallButtonControl.hallCall();
        }
}
class Dispatcher{
        update(){System.out.println("update");
                //TODO CODE HERE


                objDriveControl.desiredFloor();
        }
}
class DriveControl{
        desiredFloor(){System.out.println("desiredFloor");
                //TODO CODE HERE


                objDrive.move();
                objDrive.stop();
        }
}
```

```
class Drive{
        stop(){System.out.println("stop");
                //TODO CODE HERE


                objHallButtonControl.atFloor();
                objDoorControl.atFloor();
        }
}
class DoorControl{
        atFloor(){System.out.println("atFloor");
                //TODO CODE HERE


                objDoor.open();
                objDoor.close();
        }
}
```

Figure 4.11: Code updated by Algorithm 4.3

Algorithm 4.3 expands the method definitions in each class. The code generated from Algorithm 4.3 is shown in figure 4.11.

4.5 Analysis

We have developed a tool, AutoKodeUC, which implements the method that we explained in section 4.3. We analyzed the code generated from AutoKodeUC. It is found that it can generate class definitions and the skeletal member function definitions. Table 4.1 gives a summary of the category of code generated.

Table 4.1: Category of code generated by AutoKodeUC

Sl. No	Type of code	Generated by AutoKodeUC
1	Variable declaration & initialization	No
2	Method declarations & definitions	Yes
3	Method calls	Yes
4	Class definitions	Yes
5	Main class and Main method	No
6	Constructors	No

The code generation from the UML uses case and sequence diagram is very limited. Use cases and sequence diagram will help us to generate method declarations and definitions, class definitions and method calls. The method and class definitions remain incomplete since we won't be able to generate variable declarations & initialization and any modification of the variables. Since, neither use case nor sequence diagram gives the overall working of the system; we cannot generate main class or main method.

We have taken the elevator system as case study and analyzed the code generated from the system models (class diagram, use case and sequence diagram). 14 classes are there in the class diagram. The code generated for each class is shown in Figure 4.12. It is obvious that the code generated is proportional to the number occurrence of the class in the sequence diagram. If a class is involved in many scenarios, then it will help us to generate more lines of code. For example, Drive, DriveControl and DoorControl are the three classes which involved in the use cases and scenarios considered in the example. So the code generated for those classes are comparatively high.

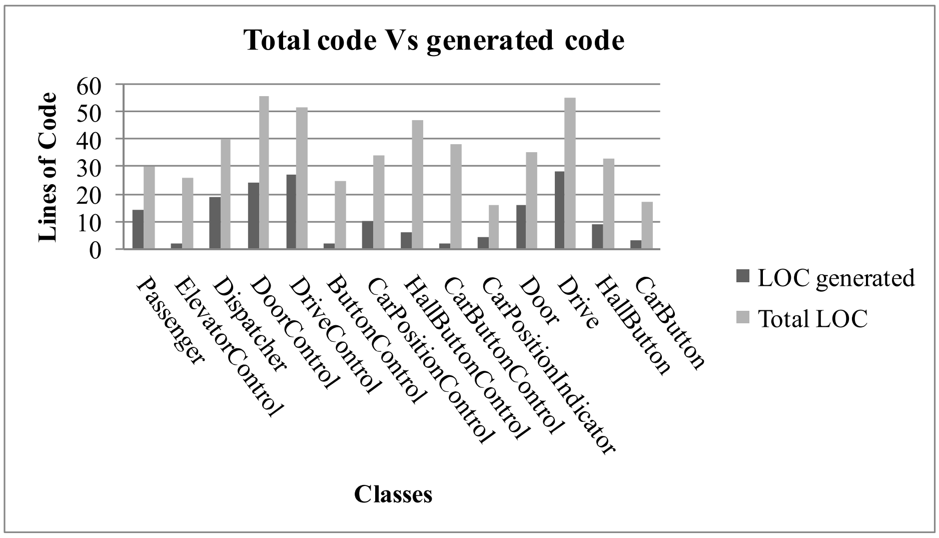


Figure. 4.12: Comparison of code generated and actual code

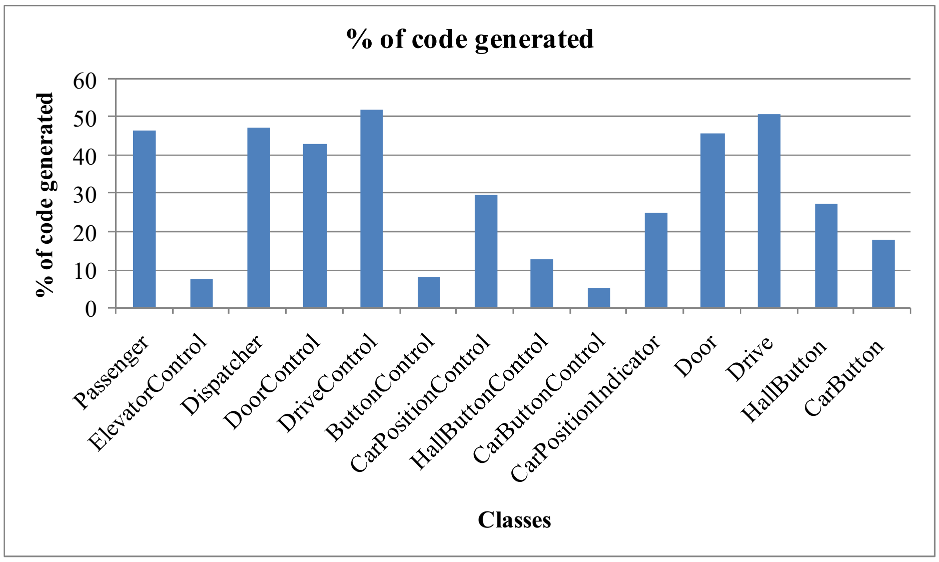


Figure 4.13 : Percentage of code generated

Figure 4.13 shows that the percentage of code generated for those classes are more than 40%. If we include more details in the sequence diagram, we can generate more code out of it. It will improve the completeness of the prototype and then improve the completeness of the requirements. On an average, this method generates 30% of total code.

Sequence diagrams help us to add details to the class definitions and the use case diagram allows us to update the actor classes.

4.6 Conclusion

The functional requirements of a system can be modeled using UML use case diagram and sequence diagrams. It is easy for the client to understand the system. For the system designers (architects), these diagrams give a baseline on what to be designed and implemented. If we are able to automatically generate code from these diagrams, it will be easy for the client as well as the system architect to communicate each other by generating prototypes. It will help us to refine the system requirements and thereby reduce the change in requirements in the later phases of software development. Moreover, we can reduce the development time and cost by the generation of such prototypes.

This chapter explained a method to generate code of a software system from the UML use-case diagram and sequence diagram. The literatures show that almost all works concentrate on sequence diagrams alone not on use case diagrams [18, 22, 30, 83, 86, 93, 102, 110, 145]. Researchers do not consider use case diagram for code generation because it does not contain any platform specific details or technical details for implementation. Use case diagrams help us to organize and define the software. So, it is good to start code generation from this model. Hence, in our work, use-case diagram is used to frame the context class and the sequence diagram is used to add details to the class.

Even though many research works [22, 30, 83, 86, 93, 102, 110, 145] focused on code generation from the sequence diagrams, a precise step by step approach is not found in the literatures. So we presented algorithms for code generation from the use case and sequence diagrams. The code generation from use case diagram is done in five steps and from sequence diagram is done in three steps. Algorithm for each one is given in the chapter. The algorithms give a formal way to do the code generation and this method is easy to implement. The analysis of the proposed method shows that it can generate even more than 30% of code for frequently interacted classes. This is a promising result.

Chapter 5 CODE GENERATION FROM ACTIVITY MODELS ENHANCED WITH OCL

5.1 Introduction

Behavioural modeling is very much important in the context of automatic code generation, since it helps us to represent the control flow in the system. Activity diagram is one of the most important diagrams for behavioural modeling. It is the only UML diagram which models control flow (work flow). State models lack this information. Activity diagram gives the activity model of the system which shows the workflow from activity to activity. Unlike state chart diagrams, activity diagrams are activity centric and it shows flow of control from activity to activity.

Activity diagram includes elements to show control flow. For example, action, activity, activity edges, swim lanes (partitions) etc, are some of the strong elements in activity diagram. The activity diagram can be considered as a graph [121].

The activity graph contains nodes and edges. The nodes can be control nodes such as initial & final nodes, decision nodes, fork node, join node etc. Edges in activity graph are the activity edge which shows the transition from one activity to another. These nodes can be converted to programming constructs without much complexity. The edges give the sequence order of the operations (or activities). The concept of activity graph helps us to traverse through the activity diagram and generate the overall execution logic of the system.

OCL expressions are used with UML models to formally specify the constraints in a precise and concise way, where the graphical notations fail to do so. It can be used for specifying pre/post conditions on operations and methods, the actual parameters that are passed to the operations, the initial values of the attributes, the guard conditions etc. It can also be used to specify invariants and types in UML class diagrams. It can be even used as navigation language and also to specify the well formedness rules of the UML meta model. The evaluation of the OCL expression [16, 17, 47] will not change the state of the system. So, it is safe to use the OCL expressions in the activity diagrams and they are said to be side-effect free.

Few research works have been reported on code generation from the UML activity diagrams, but those works primarily concentrate on control flow generation. Moreover, those works do not present any well defined formal method for code generation. Method definitions are not generated because activity diagrams show a high level activity model. Each activity in the activity diagram can be a method (function) in the program. The actions inside each activity and the control flow between them may not be specified in activity diagrams. Fine tuning the activity models to include those details will help in improved code generation.

In chapters 5 and 6, we propose different methods for this fine tuning. One method is to use OCL statements in the activity diagram which is explained in this chapter. Another approach is to expand each activity node with a sub activity diagram as we proposed in our work [105]. The third approach is to expand the activity nodes with sequence diagrams, which is explained in the next chapter.

The use of code generators will improve the software development process and it raises the quality of the code produced [4, 10, 18, 28, 51, 82, 86, 92, 105, 127, 147]. Many CASE tools are available in software development which supports UML standards for system design. Most of them won't support OCL as a specification language [7]. In this scenario, we introduce the meta models for including the OCL statements in the UML models. It will encourage the CASE tool developers to include OCL in their products and there by improve the software process to much better level.

In this chapter, we propose the meta models to include OCL expressions in the UML activity models. Different possibilities to include

OCL in activity models are explored in our previous work [119]. The operational semantics for OCL enhanced activity diagram is defined in this chapter. We also present a method to generate code from the OCL enhanced activity models. We followed the Model Driven Development (MDD) approach [97, 64] in our work, since OCL is an important component of model driven engineering. The developers who follow MDD will start with system designing using UML models and then generates the code automatically from the system designs. In MDD approach, the model compiler converts the platform independent model (PIM) which is system model in executable UML to platform specific model (PSM) like C/C++ programs [8, 15]. It prevents the alterations in PSM. MDD streamlines the software development process. It provides traceability between system requirements and system design elements. MDD even supports model execution, which will help us to locate logical errors and inconsistencies in the system specifications at early stages of the software development. It reduces the development effort.

The main contributions of this chapter:

- We have shown how to connect OCL statements for operation body, actual parameters, initial values, instances and guard conditions with the activity diagram. Meta models for the same are also provided in this chapter.
- We have developed operational semantics for OCL enhanced activity diagram and used this for proving correctness of the code generation algorithm.
- We devised a concrete method to convert the OCL enhanced activity models to source code.

5.2 Meta Model For OCL Expression In UML Activity Diagram

This section proposes a theoretical background for including OCL in UML activity models. Meta models give an excellent way to describe the models. So, in this section, the authors present the meta models for incorporating OCL statements in UML activity diagrams. It gives a clear picture on how to include OCL in activity models. The authors studied the possibilities to include OCL in UML activity diagram and examined the UML meta models [139] and OCL meta models [98] and formulated the meta models to incorporate OCL expressions [134, 68] in activity diagram.

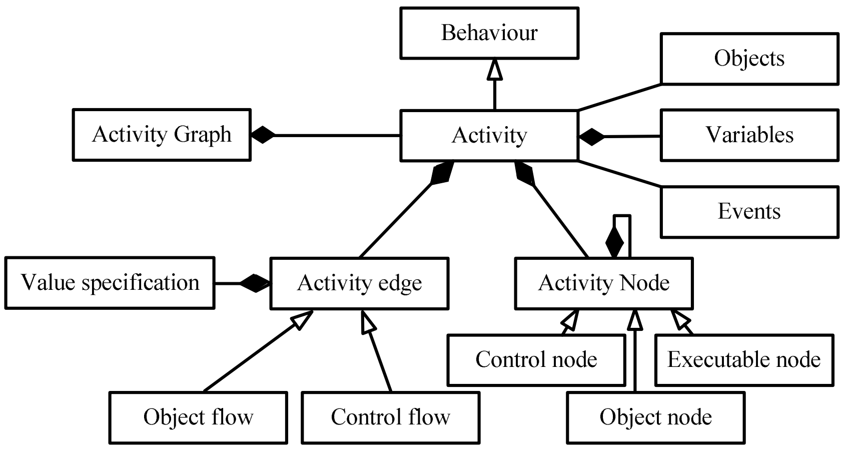


Figure 5.1: Simplified Meta model of UML2.0 Activity Diagram

5.2.1 State of the art

The OMG specification for the UML activity diagram considers an activity diagram as a graph called Activity Graph. Activity Graph consists of activities as shown in Figure 5.1. Each Activity is associated with an Object which is responsible for the activities to be done. There will be associated Events and variables. Each activity is composed of Activity Edges and activity nodes. Each activity edge can be either Control flow or Object flow. Each Activity Node can be Control node, Object node or Executable node. Control nodes are initial node, final node, decision node, merge, fork or join. Executable nodes are the directly executable action nodes. The Value Specification associated with activity edge involves guard conditions, variable/object values, etc.

The Object Management Group has defined the metamodel for OCL expressions as shown in Figure 5. 2 [98]. The shaded classes are the new additions to the UML. *ExpressionInOcl* which is defined to be a subclass of the class *Expression* in the UML meta model.

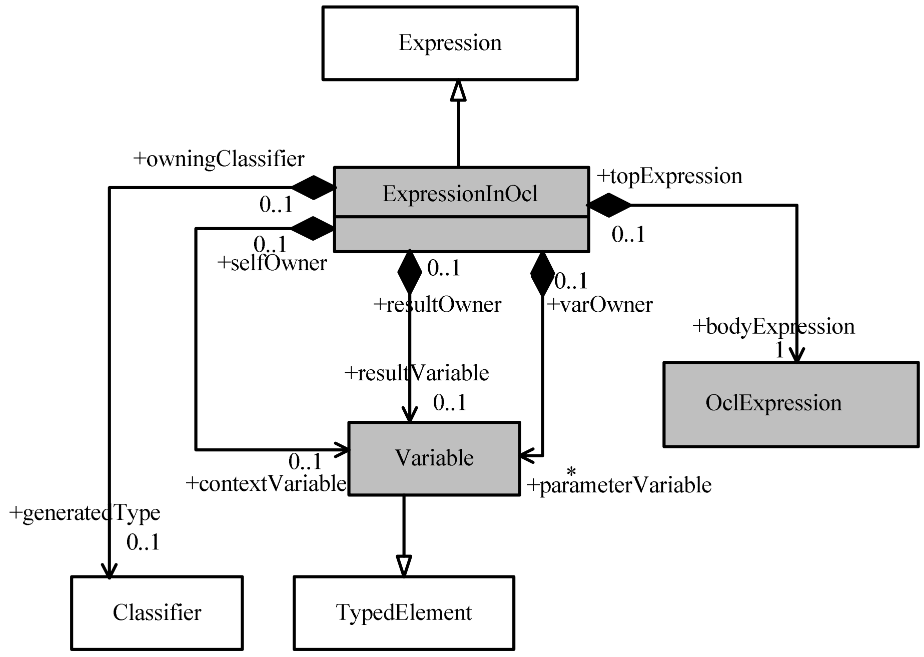


Figure 5.2: OCL in UML

OclExpression class is associated with *ExpressionInOcl* through the *bodyExpression* attribute. *ExpressionInOcl* is associated to *Variable* class through different type of variable attributes, like *contextVariable*, *parameterVariable* and *resultVariable*. The ExpressionInOcl class is used in the following meta models to represent the OCL expressions. This is the current state of the OCL specification and the UML Activity diagram specification.

In the following sub sections, the authors present the meta models to merge the models in Figure 5.1 and Figure 5.2 to incorporate the OCL statements in the activity graph. The OCL expressions can be used for specifying Operation contracts, Operation bodies, Initial values, Instances, Actual parameters and Conditions.

5.2.2 Meta model for operation contracts

The activities in an activity diagram may have some pre conditions, which should hold when the execution of the activity starts. Similarly, the activities may have post conditions which should hold when the activity completes its execution. This pre and post conditions can be specified in a UML activity diagram with the help of OCL notations [42, 41, 13, 32, 52].

The pre and post conditions are added to the UML meta model as Constraint class. The Constraint class is associated to the Action class with values *localPrecondition* and *localPostcondition* which are elements in the *ownedElement* set defined in UML2.x specification. These constraints are implemented using the OCL expressions. The Constraint class is associated to the Expression class and which is the superclass of ExpressionInOcl. The metamodel that is shown in Figure 5.3 implies that the pre and post conditions can be associated with the Activity node in the activity diagram.

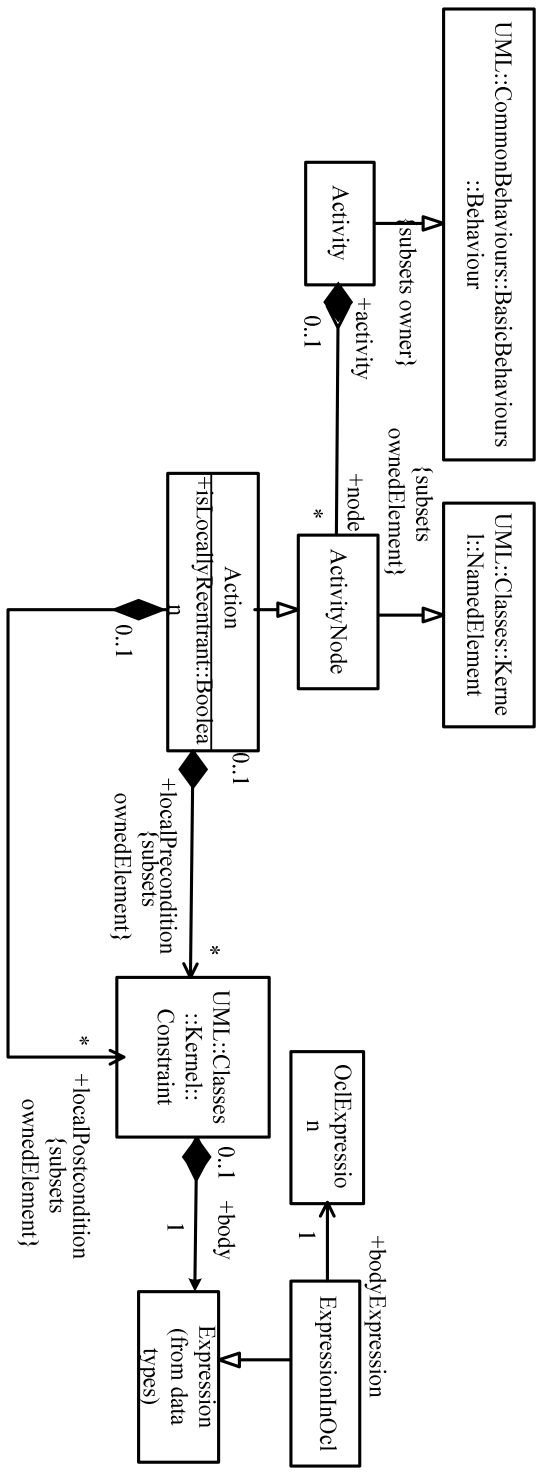

UML::CommonBehaviours::BasicBehaviours ::Behaviour
Activity
{subsets owner}
+activity
0..1
{subsets ownedElement}
+node
*
UML::Classes::Kernel::NamedElement
ActivityNode
Action
+isLocallyReentrant::Boolean
0..1
0..1
+localPrecondition {subsets ownedElement}
*
UML::Classes ::Kernel:: Constraint
*
+localPostcondition {subsets ownedElement}
OclExpression
0..1
+body
1
1
+bodyExpression
ExpressionInOcl
Expression (from data types)

Figure 5.3: Metamodel for Pre and Post Condition of an Action

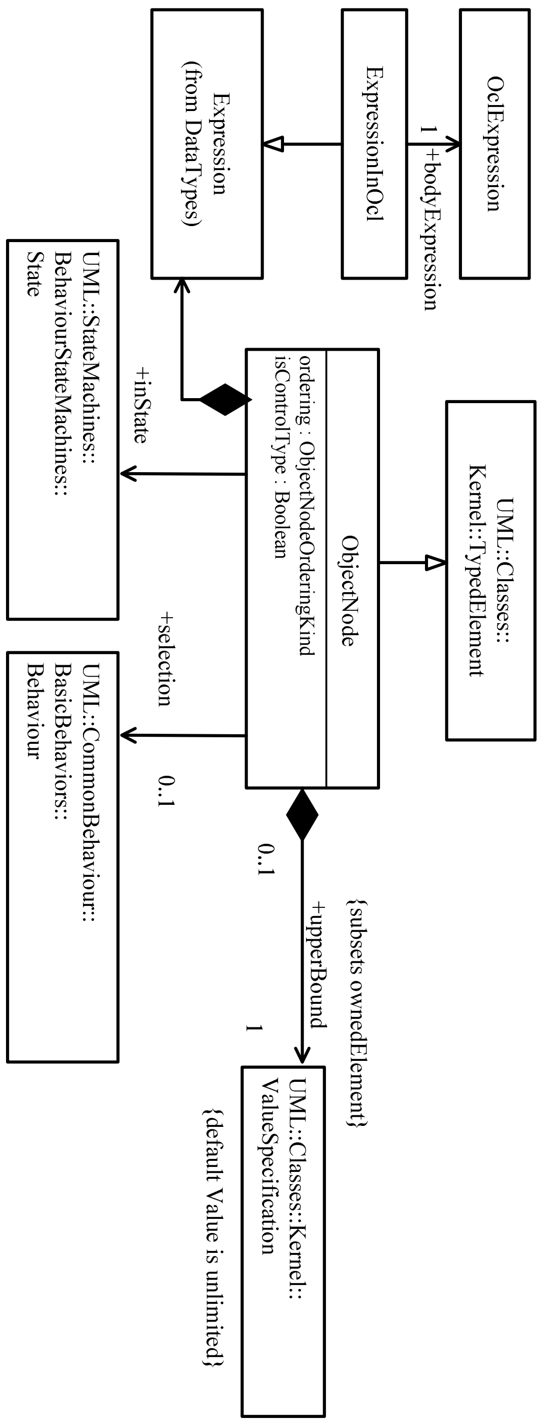

OclExpression
1 +bodyExpression
ExpressionInOcl
Expression (from DataTypes)
UML::Classes:: Kernel::TypedElement
ObjectNode
ordering : ObjectNodeOrderingKind
isControlType : Boolean
+inState
UML::StateMachines:: BehaviourStateMachines:: State
+selection
0..1
UML::CommonBehaviour:: BasicBehaviors:: Behaviour
0..1
{subsets ownedElement}
+upperBound
1
UML::Classes::Kernel:: ValueSpecification
{default Value is unlimited}

Figure 5.4 : Metamodel for OCL representation of instances (objects)

5.2.3 Meta model for initial values

OCL provides mechanisms to specify the initial values of attributes as well as association ends. The initial value representation in UML metamodel is shown in the Figure 5.4. The initial value is always attached to the attribute/property of a classifier or to an association end. The metamodel implies that the property/attribute value can be represented using an OCL expression. The *initialValue* should be of the type of the attribute. The *Property* class is associated to the *Expression* class. The association is based on the *property*.

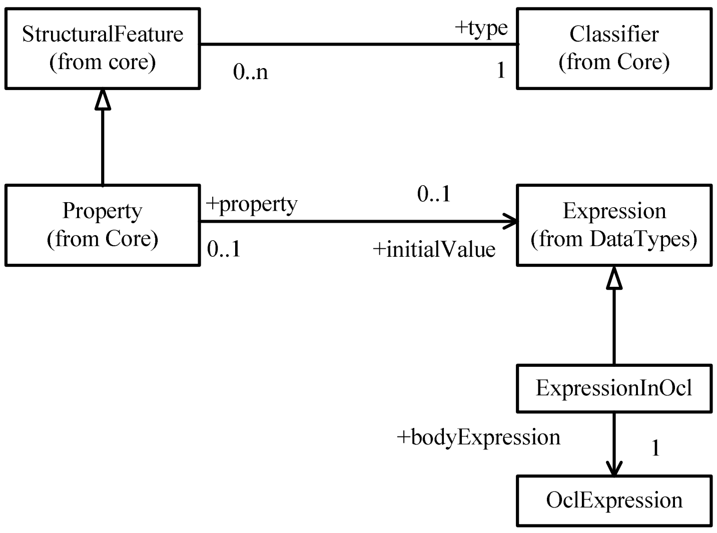


Figure 5.5 : Metamodel for OCL representation of Initial value

5.2.4 Meta model for instance

The instance (object) of a class can be identified with unique identifiers. OCL 2.x supports this unique identification of objects [101, 23].

The metamodel for instance representation using OCL is shown in Figure 5. 5. The *ObjectNode* is a subclass of *TypedElement*. The *ObjectNode* have *State* and *Behavior*. The object node can be represented using the OCL expression.

5.2.5 Meta model for actual parameter

OCL provides syntax to mention the actual parameters to an activity. The list of parameters as well as the return type of the activity (operation) can be mentioned [19].

The metamodel for including the OCL expression in parameter specification is shown in the Figure 5. 6. *Parameter* is associated with the *ParameterSet*. *ParameterSet* is a subclass of *NamedElement* as per the UML 2.x specification. Each parameter can be represented using the OCL expression. There can be once OCL expression corresponds to each *Parameter*.

5.2.6 Meta model for condition

In activity diagrams, the guard conditions (or conditions, in short) can occur in decision nodes. The guards will be given inside square brackets in the activity diagram.

Figure 5.7 shows the metamodel for the OCL representation of guard conditions of decision nodes. The *DecisionNode* class is associated to the *Guard* class using the guard condition. *DecisionNode* and other siblings are subclasses of *ControlNode* class.

Decision node has zero or one guard conditions. These guard conditions are represented using the OCL expression. The same method can be applied to the other control nodes like, *JoinNode*, *MergeNode* etc.

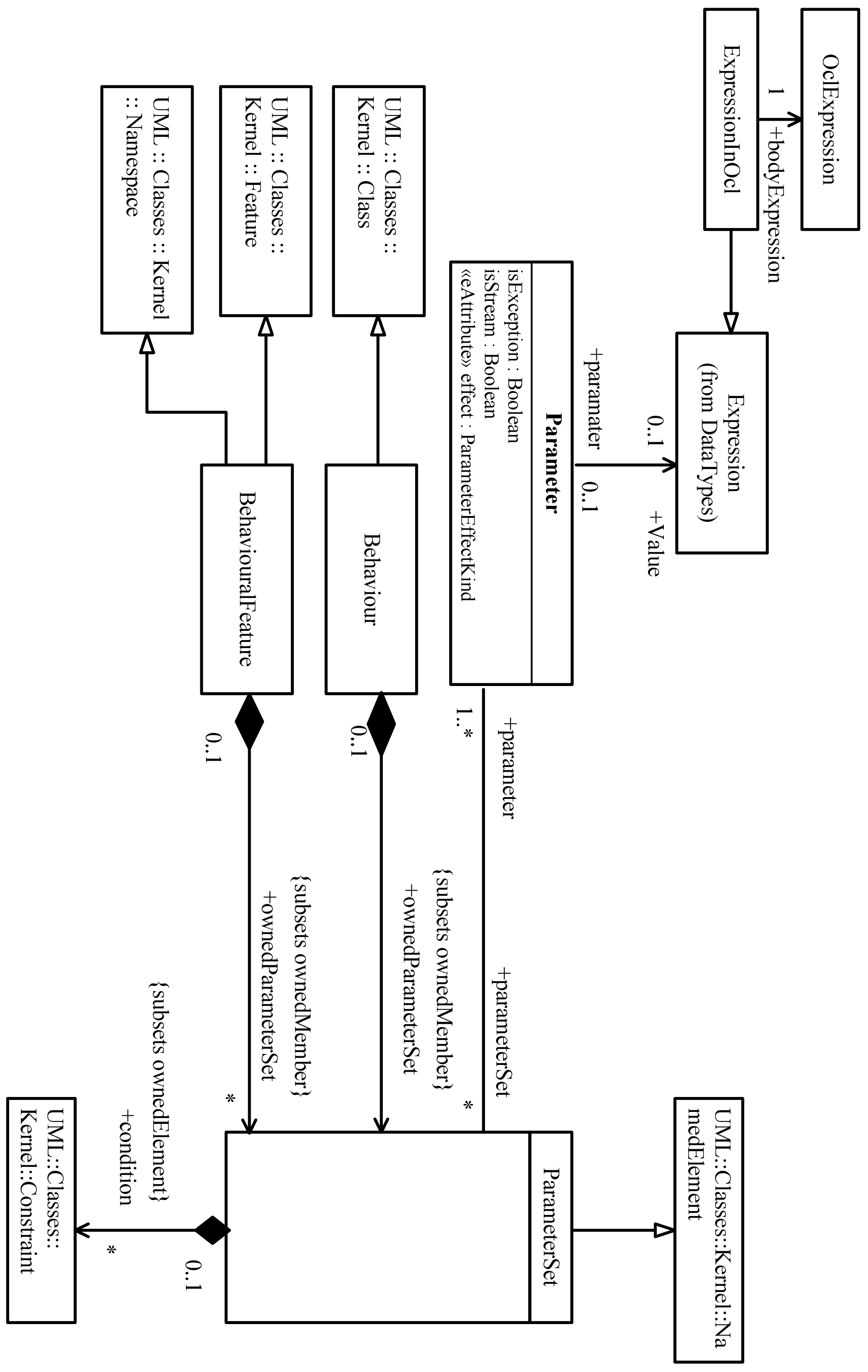


Figure 5.6: Metamodel for Actual parameter to an action

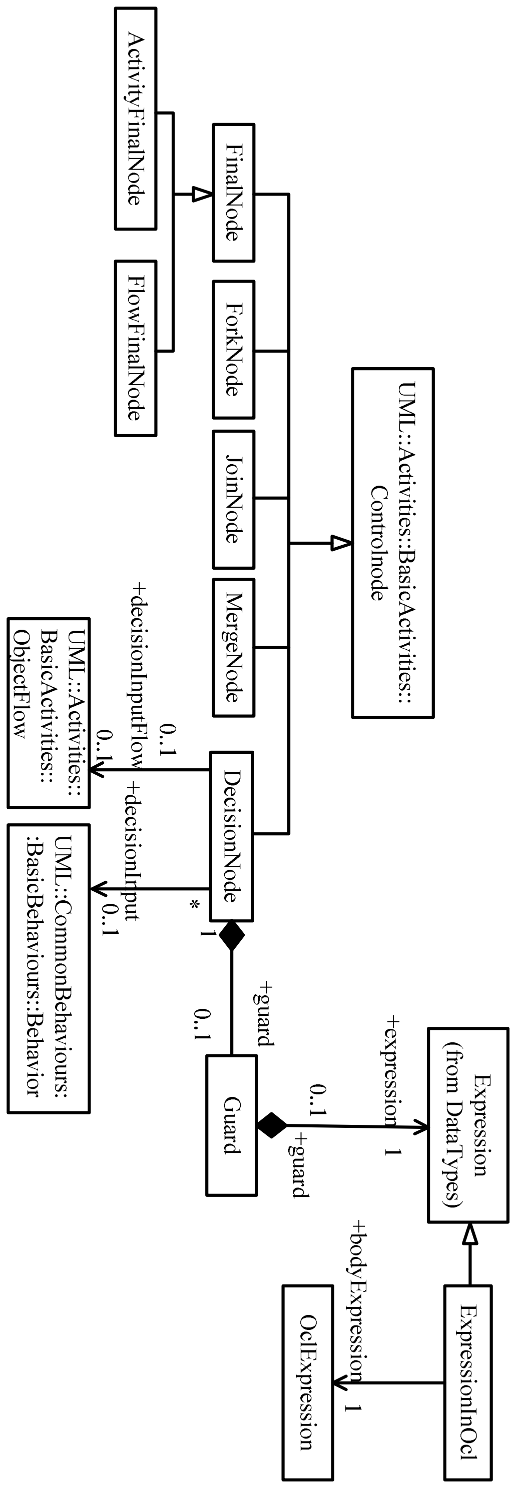


Figure 5.7: Meta model for OCL representation of guard conditions of decision node

5.3 Formal Semantics of OCL Enhanced Activity Diagrams

AG_{ocl} = (N, E, e, G, var, o, ψ), where
N = { x | x can be of type AN, CN, EN or ON}
E = {(x,y) | transition from x to y} where (x and y) ∈ N
e= {x | x is an external event}
G = {x | x is a guard expression}
var = {x | x is a local variable}
o= {x | x is an object or actor}
ψ = (C, v, t, Φ) which represents the OCL expressions applied in the activity graph.
Where C - the classifier or the context,
v = {v_c, v_r, v_p},
t – the type of the OCL statement and
Φ – OCL statements

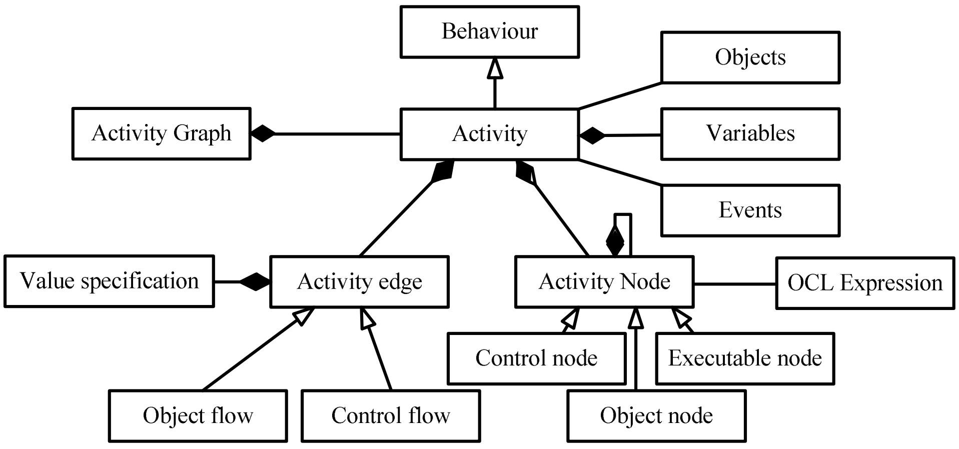
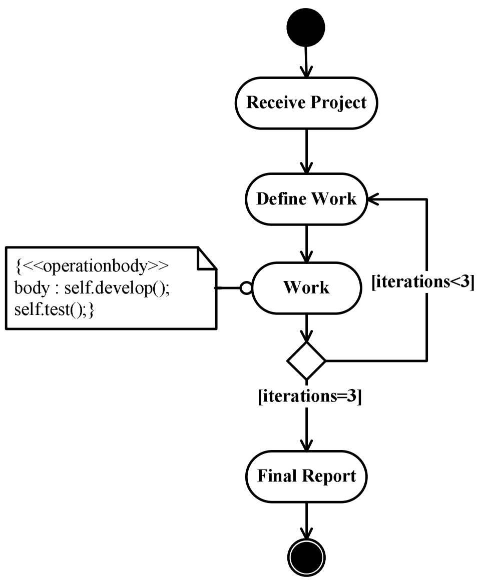


Figure 5.8: Simplified Meta model of UML2.0 Activity Diagram with OCL expressions

An Activity Graph, AG_{ocl}, is a tuple which contains nodes N, edges E, events e, guard conditions G, local variables var, set of objects o and OCL expressions ψ. A node can be of four types, AN (*activity node*), CN (*control node),* EN *(executable node) and* ON *(object node)*. AN is the activity node which represents activity in AG_{ocl}. CN represents the control node which includes the decision node, fork, and join. EN is the executable node which represents single atomic action in AG_{ocl}. E is the transition from one node to another, where the nodes ∈ N. 'e' is the external events that can occur in the system. '$\mathcal{G}$' is the guard expressions that can be applied to edges. Ψ is the OCL expression that can be used in the activity graph.

The OCL expressions include the parameter values, initial values, instances, conditions, operation contracts etc. An OCL expression may contain the classifier (C), context variables (v_c), parameter variables (v_p), result variables (v_r) and OCL statements (Φ). The simplified metamodel for the OCL enhanced activity diagram is given in Figure 5. 8.



$\mathcal{AG}_{ocl} = (N, E, e, \mathcal{G}, var, \sigma, \psi)$, where

N = {n1, n2, n3, n4, n5, n6, n7}

E = {(n1, n2), (n2, n3), (n3, n4), (n4, n5),
(n5, n3), (n5, n6), (n6, n7)}

e = { }

$\mathcal{G}$ = {iterations=3, iterations<3 }

var = {iterations}

σ = {obj_pd_1}

ψ = (ProjectDevelopment, {iterations},
{operationbody}, {self.develop();self.test();})

Figure 5.9 : The activity diagram Project Development and its formal semantics

Figure 5.9 gives the activity diagram that represents the control flow of Project Development. The formal semantics of it is given in Figure 5.9 and Figure 5. 10. Project Development has four activity nodes, one decision node, initial node and final node. The nodes have been named internally as n1, n2, .. n7, including pseudo nodes and activity nodes. The edges (E) have been defined as a pair of end nodes. First edge is from initial node to 'receiveproject'.

That's from n1 to n2. So the edge is represented as (n1,n2). The guard conditions G are given as iterations=3 and iterations<3. The activity node 'work' is enhanced with OCL statements for the operation body. The variables (var) used in the control flow is 'iterations'. The object (o) which is active is obj_pd_1. The OCL expression (ψ = (C, v, t, Φ)) has the class ProjectDevelopment as context (C), *iterations* as local variable (v), type (t) of the OCL statement is operationbody and the statements (Φ) are '*self.develop(); self.test();*'.

The operational semantics of this activity diagram is shown in Figure 5.10. The variable definition and initialization is given. The variables are *activeNode*, *ac* and *iterations* and seven *ID* variables correspond to each node in the activity diagram. In the following section the variables have been initialized. What happens when edges are taken is explained in the '*definition'* section. The rules for state transition are described in the '*transition'* section.

5.3.1 Operational semantics of OCL enhanced activity diagrams

We define operational semantics of OCL enhanced Activity Diagrams using Finite State machine (FSM) by extending the semantics given in [112]. State variables and a set of predicates describing the transitions of the state variables are given in the description of the FSM. The state variable(s) will be changed at each state. The transition functions described using the predicates give the relation between current value and next value of the state variables. The logical operators &, | and ! are used in the predicates. Constant values 1 and 0 are used to denote true and false respectively.

An Activity Diagram is mapped into FSM. The execution of a single activity in the activity diagram is mapped as a single step in the FSM. The final node of the activity diagram is mapped as infinite loop of "no operation" action.

5.3.2 Variables used in the finite state machine

The state variables have been defined in Rule 1. First one is the variable ($in\_an.nID$) to represent the control flow in each node. This variable ($in\_an.nID$) says whether a node is active or not. The control nodes like Fork and Join nodes have separate variables for each outgoing ($in\_Ft.tgt.nId$) transition and incoming ($in\_Jt.src.nId$) transitions respectively. These variables are used to decide which transition steps can be executed. 'o' is the set of objects that take part in the process flow. Each node in the *AG* belongs to one of these objects. (i.e, $an.obj \in o$).

The variable *activeNode* denotes the activity node which is currently active. The variable *ac* holds the name of the executed action. The unique id of each node is denoted as nID.

The initialization of the state variables is given in Rule 2. The variable of the initial node ($in\_ad.initialNode.nID$) is set to true and other variables are set to false. Local variables (*v*) are initialized to their pre-defined values. The value of *activeNode* and *ac* is determined by the initial node.

Rule 1: Variables

$$\begin{aligned}
&\forall an \in AN \cup CN_{init,final} \wedge an.obj \in o :\ in\_an.nId : boolean;\\
&\forall fn \in CN_{fork} \forall t \in fn.out \wedge fn.obj \in o : in\_Ft.tgt.nId : boolean;\\
&\forall jn \in CN_{join} \forall t \in jn.in \wedge jn.obj \in o :\ in\_Jt.src.nId : boolean;\\
&activeNode : \{\textstyle\bigcup_{an \in AN} an.nId\};\\
&ac : \left\{\cup_{acname \in \{N.name\}} acname\right\};\\
&\forall_{\mathrm{v}} \in \mathrm{var} :\ \mathrm{v}.name : \mathrm{v}.typeDecl;
\end{aligned}$$

Rule 2: Initialization

$$
\begin{aligned}
& in\_ad.initialNode.nID = 1\ \& \\
& \forall an \in AN : in\_an.nId = 0\ \& \\
& \forall fn \in CN_{fork}\ \forall t \in fn.out : \wedge\, in\_Ft.tgt.nId = 0\ \& \\
& \forall jn \in CN_{join} \forall t \in jn.in : \wedge\, in\_Jt.src.nId = 0\ \& \\
& \forall \mathrm{v} \in \mathrm{var} :\ \mathrm{v}.name = \mathrm{v}.init\ \& \\
& activeNode = ad.initialNode.nID\ \& \\
& ac = ad.initialNode.acName;
\end{aligned}
$$

Rule 3: Definition of Transitions

3.1 $\forall t \in E \wedge t.src, t.tgt \in AN \cup CN_{initial,final}$:

$$
\begin{aligned}
& t\_taken := in\_t.src.nID\ \&\ \vee_{t.src \in AN}\ in\_t.src.\psi \\
& \&\ \vee_{t.tgt \in AN}\ next(in\_t.src.\psi)\ \& \\
& \vee_{t.src \in AN} in\_t.src.e\ \&\ \vee_{t.tgt \in AN} next(in\_t.src.e) \\
& \&\ \mathrm{next(in\_t.tgt.nId)}\ \& next\left(activeNode = t.tgt.nID\right);
\end{aligned}
$$

3.2 $\forall t \in E \wedge t.src \in CN_{fork}$:

$$
\begin{aligned}
& t\_taken := in\_Ft.tgt.nId\ \&\ \mathrm{next(in\_t.tgt.nId)} \\
& \&\ next\left(activeNode = t.tgt.nID\right);
\end{aligned}
$$

3.3 $\forall t \in E \wedge t.src \in CN_{join}$:

$$
\begin{aligned}
& t\_taken := \wedge_{t \in t.src.in}\ in\_Jt.src.nId\ \&\ \mathrm{next(in\_t.tgt.nId)} \\
& \& next\left(activeNode = t.tgt.nID\right);
\end{aligned}
$$

3.4 $\forall t \in E \wedge t.tgt \in CN_{merge}$:

$$
\begin{aligned}
& t\_taken := in\_t.src.nID\ \&\ \mathrm{next(in\_t.tgt.nId)} \\
& \&\ next\left(activeNode = t.tgt.nID\right);
\end{aligned}
$$

3.5 $\forall t \in E \wedge t.src \in CN_{decision}$:

$$
\begin{aligned}
& t\_taken := in\_t.src.nID\ \&\ t.G\ \&\ \mathrm{next(in\_t.tgt.nId)} \\
& \&\ next\left(activeNode = t.tgt.nID\right);
\end{aligned}
$$

The edges in the activity diagram are taken as the transitions in FSM. Rules 3.1 to 3.5 specify the possible transitions. The transitions from and to the activity nodes, initial node, final node, fork, join, merge and decision nodes are considered. In each transition the *t_taken* and the *activeNode* variables are updated according to the type of source and target nodes of the edge.

The next() function is used to change the value of the variables. The next() function assigns the next state value to a variable based on the current state value of the variables [14]. The next() function is used to evaluate statements or Boolean expressions at the next state of the system.

In rule 3.1, the transition from or to the initial node ($CN_{initial}$), final node (CN_{final}) and activity node (AN) is considered. When a transition is taken, the variable *t_taken* enables successor control flow variables *(in_t.tgt.nId)* in the next state and the variable *activeNode* is updated to the target node of t (*t.tgt.nID*). If the *activeNode* is an *AcceptEvent* action node (i.e, $in\_t.src.e$ is true), the corresponding target node (*t.tgt.nID*) will be taken as the next *activeNode*.

Rule 3.2 states that every transition t leaving a fork node (CN_{fork}) can be taken if it's control flow variable *in_Ft.tgt.nId* is true. These variables are set in rule 4. Rule 3.3 states that, to take a transition preceding a join node, all control flow variables *in_Jt.src.nId* have to be true, indicating that all previous concurrent branches have reached the join node.

The variables *in_Ft.tgt.nId* and *in_Jt.src.nId* controls the transition to and from the fork and join nodes respectively. Rule 3.4 specifies that, for merge nodes (CN_{merge}), the incoming control flows are routed to the outgoing edge. In the case of decision nodes ($CN_{decision}$), the transition will be taken if the guard condition ($t.G$) of the outgoing edge is true. It is given in Rule 3.5.

$Rule\,4$

$$\forall fn \in CN_{fork}:$$
$$(activeNode = fn.nodeID -> \wedge_{t \in fn.out}\, next\left(in\_Ft.tgt.nId\right))\ \&$$
$$\forall jn \in CN_{join} \forall t \in jn.in:$$
$$\wedge\,(activeNode = jn.nodeID -> next\left(in\_Jt.src.nId\right));$$

$Rule\,5$

$$\forall n \in AN:$$
$$((in\_n.\psi -> execute(n.\psi))\ \&$$
$$in\_n.nID = next\left(in\_n.nID\right) \Big|$$
$$\vee_{t \in n.in. \cup n.out, t\_taken\,defined}\, t\_taken \Big|$$
$$\vee_{t \in n.out.tgt.out, t.src \in CN_{fork}}\, t\_taken \Big|$$
$$\vee_{t = n.out.tgt.out, t.src \in CN_{join}}\, t\_taken\ \Big)\ \&$$
$$\forall fn \in CN_{fork} \forall t \in fn.out:$$
$$\left(in\_Ft.tgt.nId = next\left(in\_Ft.tgt.nId\right)\Big|\right.$$
$$\left.\vee_{t' \in fn.in \cup t}\, t'\_taken\right)\ \&$$
$$\forall jn \in CN_{join} \forall t \in jn.in:$$
$$\left(in\_Jt.src.nId = next\left(in\_Jt.src.nId\right)\Big|\right.$$
$$\left.\vee_{t' \in t \cup jn.out}\, t'\_taken\right);$$

$Rule\,6$

$$\forall n \in CN_{final}:$$
$$\wedge\, in\_n.nID -> next\left(activeNode = nop\right)\ \&$$
$$\left(\vee\, in\_n.nID \middle| \vee_{t \in T, t\_taken\,defined}\, t\_taken\right);$$

$Rule\,7$

$$\forall v \in \mathrm{var}:$$
$$\wedge\left(v.vName = next\left(v.vName\right)\ \Big|\right.$$
$$\vee_{n \in a\,\mathrm{sgn}\,Var(v)}\left(next\left(activeNode\right) = n.nID\ \&\right.$$
$$\left.\left. next\left(v.vName\right) = n.asgmtv.val\right)\ \right);$$

$Rule\,8$

$$\wedge_{an \in AN}\left(next\left(activeNode\right) = an.nID -> \right.$$
$$\left. next\left(ac\right) = an.acName\right);$$

5.3.3 Transitions

Rule 4: The fork node variables get activated when the control flow reaches the fork node. Similarly, join node variable gets activated when the control flow reaches the join node.

Rule 5: when the incoming or outgoing edges are taken, the corresponding *an.nID* will be changed. Fork nodes can change these variables with every outgoing transition and join nodes with their one outgoing transition. If the *AN* contains an OCL expression (ψ), it will be evaluated using the function execute( ). If the *activeNode* contains an OCL expression ($n.\psi$), depending upon the type of the OCL statement ($n.\psi.t$), corresponding actions will be taken by executing the OCL expressions. If $n.\psi.t$ *=initialvalue*, the variable specified will be assigned with the value given. Similarly, if $n.\psi.t$ *=precondition* or *postcondition*, those conditions will be checked.

In the case of fork (CN_{fork}) and join nodes (CN_{join}), this variable will be changed whenever an outgoing edge is taken. The fork variable *in_Ft.tgt.nId* will be changed when an incoming edge of the fork node is taken. The join variables *in_Jt.src.nId* will be changed if the outgoing edge of the join node is taken.

Rule 6: In each step a new edge will be taken till the last node.

Rule 7: The value of the local variables (*v*) changes whenever there is an assignment statement (*n.asgmtv*) in the currently executed node (*n*). The value of the assignment (*n.asgmtv.val*) will be assigned to the variable *v.vName*.

Rule 8: Assigns the value of activeNode to the variable ac.

Variables:-

```
activeNode : { n1, n2, n3, n4, n5, n6, n7, nop };
ac : { receiveproject, definework , work, decision, finalreport, nop}
iterations = {0,1,2,3,4}
nID_n1=boolean; nID_n2= boolean; nID_n3= boolean; nID_n4= boolean;
nID_n5= boolean; nID_n6= boolean; nID_n7= boolean;
```

Initialization:-

```
nID_n1=1; nID_n2=0; nID_n3=0; nID_n4=0; nID_n5=0; nID_n6=0; nID_n7=0;
iterations =  0;
activeNode=n1;
ac=nop;
```

Definition:-

```
t_taken_n1n2 = n1;
next(iterations)=0;
next(activeNode)=n2;

t_taken_n2n3 = n2
next (activeNode)=n3

t_taken_n3n4 = n3
next (activeNode)=n4

t_taken_n4n5 = n4 & (operationbody)
next(iterations)=iterations+1;
next(activeNode)=n5

t_taken_n5n3 = n5 & (iterations<3)
next(activeNode)=n3

t_taken_n5n6 = n5 & (iterations==3)
next(activeNode)=n6

t_taken_n6n7 = n6
next(activeNode)=n7
```

Transition:-

```
((nID_n1=next(nID_n1)) | t_taken_n1n2) &
((nID_n2=next(nID_n2)) | t_taken_n1n2 | t_taken_n2n3) &
((nID_n3=next(nID_n3)) | t_taken_n4n3 | t_taken_n2n3 | t_taken_n3n4 | t_taken_n5n3) &
((nID_n4=next(nID_n4)) | t_taken_n3n4 | t_taken_n4n5) &
((nID_n5=next(nID_n5)) | t_taken_n4n5 | t_taken_n5n3 | t_taken_n5n6) &
((nID_n6=next(nID_n6)) | t_taken_n5n6 | t_taken_n6n7) &
((nID_n7=next(nID_n7)) | t_taken_n6n7);
```

Figure 5.10 : The declarations and definition of transitions of the activity diagram Project Development

5.4 Code Generation From OCL Enhanced Activity Diagram

In this section we explain the code generation process from the OCL enhanced activity diagram. The OCL expressions added in the design will be checked for errors. If there is no error it will be incorporated with the XML document for code generation. The code generation steps are given in Figure 5. 11.

The system design is prepared in activity diagram (AD) and additional details, like method body, pre- and post conditions, are added with the help of OCL expressions. This system model is then converted to XML format. The UML modeler, which is used as a part of the code generator, will do the conversion. The XML follows the DTD (Document Type Definition) which is mentioned in [106]. The OCL expressions are added to the XML document as a separate element named *<OCL>* associated with node element and edge elements. When we add precondition to an activity using OCL expression, the type attribute of <OCL> will be given as "precondition". Similarly, for post condition, type is "postcondition". See Figure 5.12 for the tree view of the document. Before code generation we need to check the OCL expressions for errors. If there is any error, report it and allow the system architect to modify the design. If there is no error, rebuild the XML document and pass it to the code generation module.

The XML document, after checking the OCL expressions, will be given for code generation. First the code generator creates a java package with the name of the activity diagram. Then it searches on AG to visit all nodes in the activity graph. A method declaration will be added whenever a new action node is taken. It checks each node in the AG and identifies all action nodes and writes the method declaration for all those action nodes. The algorithm identifies currently active object and its type.

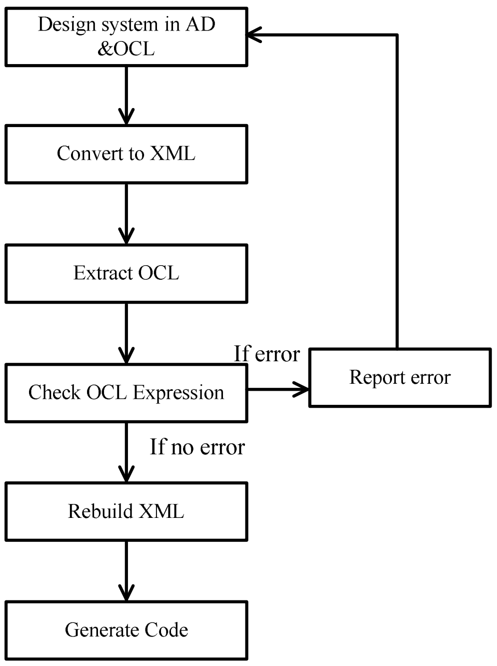
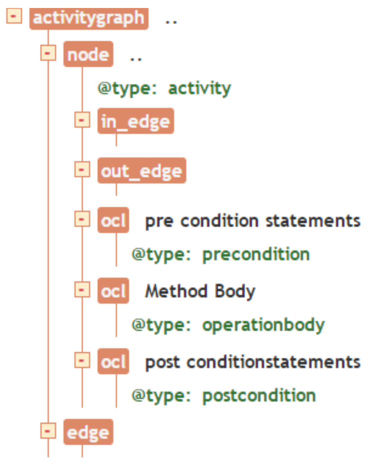


Figure 5.11: Steps to generate code from UML diagram

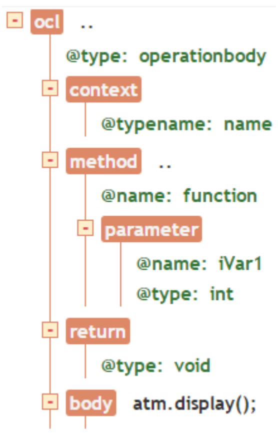
The active objects will be fetched from the XML document of AG. If no class exists corresponding to the object, then a class will be created. Otherwise the existing class will be updated with the method declarations and added to the source code SC. The search stops when all nodes have been taken. A main class will be created and embed a main() method in it.

Each activity will appear as a function call in the implementation code. So, first the XML document is parsed to get the object tree of the XML document [80]. We consider six types of nodes in the activity diagram. *Initial* node, *final* node, *activity* node, *decision* node, *fork* node and *join* node. Read each *node* element. If it is an activity node, add the operation call statement. If it contains the OCL element, the algorithm methodDefinition() has to be called. The pre and post conditions will be added to the method definitions. If the node type is *decision*, a decision making statement will be added to the implementation code. The fork node has to be handled with special care. The implementation of the fork node is the most complicated one to implement, since it handles concurrency.



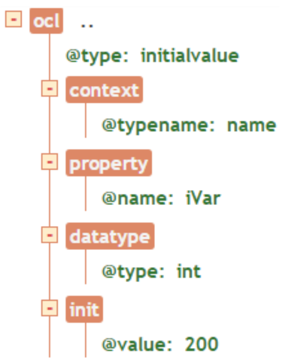
```
activitygraph ..
  node ..
    @type: activity
    in_edge
    out_edge
    ocl   pre condition statements
      @type: precondition
    ocl   Method Body
      @type: operationbody
    ocl   post conditionstatements
      @type: postcondition
  edge
```

Figure 5.12 : XML document format



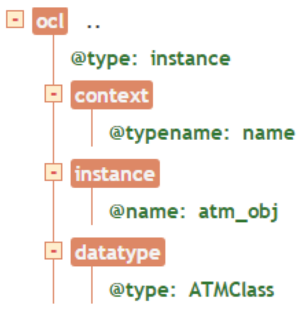
```
ocl ..
  @type: operationbody
  context
    @typename: name
  method ..
    @name: function
    parameter
      @name: iVar1
      @type: int
  return
    @type: void
  body   atm.display();
```

Figure 5.13 : Operation body in XML



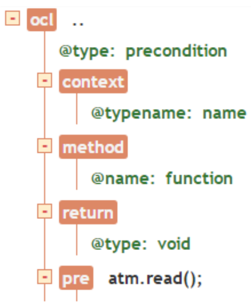
```
ocl ..
  @type: initialvalue
  context
    @typename: name
  property
    @name: iVar
  datatype
    @type: int
  init
    @value: 200
```

Figure 5.14: Initial value in XML



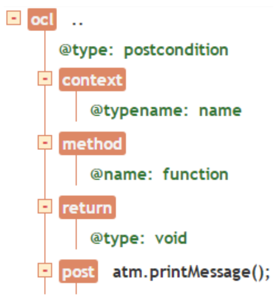
```
ocl ..
  @type: instance
  context
    @typename: name
  instance
    @name: atm_obj
  datatype
    @type: ATMClass
```

Figure 5.15: instance in XML



```
ocl ..
  @type: precondition
  context
    @typename: name
  method
    @name: function
  return
    @type: void
  pre   atm.read();
```

Figure 5.16: Pre condition in XML



```
ocl ..
  @type: postcondition
  context
    @typename: name
  method
    @name: function
  return
    @type: void
  post   atm.printMessage();
```

Figure 5.17: Post condition in XML

In our method, we implement concurrency using Threads. The number of child nodes of the fork node shows the number of concurrent paths in the process flow. There can be multiple paths between fork-join combinations. One of the paths is taken as the main thread itself. So, other paths should initiate threads. All these threads, including the main thread, have to wait at join node till all other threads reach join node. If the node type is *fork*, we have to initiate threads depending on the number of child nodes of the fork node. If there are 3 child nodes, it means 1 main thread and 2 sub threads. So we have to initiate two sub threads. Each path from fork node to join node is considered as an activity diagram. If node type is *join*, the main thread will wait for other threads to complete. Continue this process till we find the final node. These steps have been summarized in *Algorithm 5.1: codeGeneration*

Algorithm 5.1 *CodeGeneration(*AG_{ocl}, P, parentThread, currentThread, start*)*

```
Input    :   AG_ocl, P, parentThread, currentThread, start
Output   :   Updated P

Perform Depth First Search on AG_ocl by keeping start as starting vertex
1.visit next node N.out_edge.target
2.identify currently active object, ca_obj,
3.if(N.type = = activity)
        3.1 create statement ca_obj.actionName();
                write to main() if currentThread='main'
                otherwise write to run() method where
            threadname=currentThread
        3.2  If this node element contains OCL element <ocl>
                3.2.1 Call methodDefinition(P,ocl)

4. else if(N.type = = decision)
        4.1 For each N.out_edge, create "if(N.out_edge guard)" statement
                write to main() if currentThread='main'
                otherwise write to run() method where
            threadname=currentThread
5.else if(N.type = = fork)
        5.1.    count the number of child nodes, say c.
        5.2.    while ( c>=1 )
            5.2.1.      create and start thread with name parentThread _t_c ,
                        write to main() if currentThread='main'
                        otherwise write to run() method where
                threadname=currentThread
            5.2.2.      push parentThread_t_c to stack
            5.2.3.      decrement c by 1
         5.3. if (currentThread = 'main')
           5.3.1.  Call CodeGeneration(AG_ocl,P, currentThread,
                   currentThread, currentForkNode)
           5.3.2.  Pop sub threads from stack and call
                   CodeGeneration(AG_ocl,P, currentThread,parentThread_t_c,
                   currentForkNode) till stack is empty
        5.4.Else if(currentThread ≠ 'main')
            5.4.1. Call CodeGeneration($\mathcal{AG}_{ocl}$,P, currentThread,
                   currentThread, currentForkNode)
            5.4.2. modify run() method in Thread class
            5.4.3. add "if(threadname==parentThread_$t_c$){ " statement to run()
                   method
            5.4.4. call
                   CodeGeneration($\mathcal{AG}_{ocl}$,P,currentThread,parentThread_$t_c$,cur
                   rentForkNode)
6. else if ($\mathcal{N}$.type = = join)
        6.1.   if(currentThread=='main')
               return
        6.2.   else if(currentThread==parentThread)
               return
        6.3.   else if(parentThread=='main')
               add "currentThread.join()" to main() method
               return
        6.4.   else if currentThread ≠ parentThread
               add "currentThread.join()" to run() method where
               threadname=parentThread
               return
7.else if ($\mathcal{N}$.type = = merge)
        go to step 1
8.else if($\mathcal{N}$.type = = finalNode)
        return P
9. Repeat steps 1 to 7 until all nodes have been taken
10.     return
```

The method definitions are generated using the algorithm *methodDefinition*. The OCL expressions are used for this purpose. As per section 5.2, the OCL statements can appear in an activity diagram in five forms; initial values for variables, the instances, operation body, pre- and post conditions. The XML format for these data is shown in Figure 5.13, Figure 5.14, Figure 5.15, Figure 5.16 and Figure 5.17. The OCL statements in the activity diagram will be converted to XML and stored as part of the node element of *ActivityGraph* in the XML document. The <ocl> element can be of the above five types. This type is mentioned in the attribute *type*. The <ocl> element has the data *context* which gives the name of the context class.

Figure 5.13 represents the components in the OCL statements for operation body and the tags used for each component. Similarly, Figure 5.14 gives the XML tags to store the initial value of a variable. Figure 5.15 gives the XML format for storing instances that is being used in the activity model. Figure 5.16 & Figure 5.17 show the XML representation of pre and post conditions respectively. The representation of actual parameter is contained in the operation body itself. So, no separate <ocl> tag is used to store the actual parameters.

The element <ocl type=”initialvalue”> contains the variable name, variable type and the initial value. This is sufficient to generate code for the initialization of a local variable. The element <ocl type=”instance”> has the instance name and type.

These details are used for the generation of object declaration statement. The element <ocl type=”precondition”> stores the details of context name (class name), method name, return type and the precondition of the method. The class name and method name defines the scope of the precondition.

The pre conditions will be added as the assertion statements at the beginning of the method definition [31, 69, 25, 122, 71, 82, 36, 65]. Similarly, <ocl type=”postcondition”> includes the same details as that of <ocl type=”precondition”>. The only difference is that, instead of precondition it contains post conditions. This post condition will be added as the assertion statements at the end of the method definition. The method will return with the result only when the post condition is true.

When we represent operation body in <ocl>, it will contain the method name, list of parameters with their data types; return type of the method and the method body. These data will be extracted and inserted to the source code, P, in the form of method definition. If we represent initial value of a variable in <ocl>, it will contain the property (variable) name, data type and its initial value. It will be written to P as "data_type variable_name = initial_value;". In the instance declaration,the <ocl> contains the instance name and its class name. The instance declaration will be added to source code as "class_name instance_name = new class_name ( ); ".

These conversions have been formally specified in *Algorithm 5.2: methodDefinition*.

Algorithm 5.2 methodDefinition(P, ocl)

```
Input   :  P, ocl
Output :  P

1.   If ocl@type == initialvalue
        1.1 Let varName← ocl.property@name
        1.2 Let varType ← ocl.datatype@type
        1.3 Let varValue← ocl.init@value
        1.4 Add the statement to P  "<varType>  <varName> = <varValue>;"
2  If ocl@type == instance
        2.1 Let insName← ocl.instance@name
        2.2 Let insType← ocl.datatype@type
        2.3 Add statement "<insType> <insName> = new <insType>( );" to P.
3  If ocl@type== precondition
        3.1 Open method with name ocl.method@name in P.
        3.2 Let preCondition← valueof(ocl.pre)
        3.3 Add the 'if condition'
            " if(<preCondition>==true){ " to the opened method immediately
            after the opening {
4  If ocl@type== postcondition
```

```
        4.1 Open method with name ocl.method@name in P.
        4.2 Let postCondition← valueof(ocl.post)
        4.3 Add 'if condition'
            "if(<postCondition>==true) return result;" to the opened method
            just before the closing }
5   If ocl@type == operationbody
        5.1 Let returnType← ocl.return@type
        5.2 Let metName← ocl.method@name
        5.3 Let i=1;
                5.3.1   Let param<i> ← ocl.method.parameter@name
                5.3.2   Let paramT<i> ← ocl.method.parameter@type
                5.3.3   i=i+1;
                5.3.4   repeat steps 5.3.1 to 5.3.3 until there is no more
                        parameter to read from ocl.method.
        5.4 Create method signature as "<returnType> < metName>
            (<paramT1> <param1> , <paramT2> <param2>, …
            <paramT<i>> <param<i>>) {" and add to P
        5.5 Add method body as "<valueof(ocl.body)> }}" to P
6   Return P
```

5.5 Proof of Correctness of the Algorithm

The operation semantics of AG gives the definition of transitions from activity nodes, initial node, final node, fork node, join node, merge node and decision node. The algorithm 5.1 CodeGeneration( ) illustrates how these nodes and transitions are converted to source code. The algorithm perform depth first search on AG starting with the initial node. It generates source code corresponds to each node and edge present in the AG. The algorithm processes the nodes of the type activity, decision, merge, initial, final, fork and join. If the AG is enhanced with the OCL statements, that will also be converted to source code.

The operation semantics gives the behavior of AG mainly focusing on when to take a transition and which is the next node to be visited. First part of the semantics (Rules 1 and 2) gives the variables used and their initialization. Second part (Rules 3.1 to 3.5) is the definition of transitions. The third part (Rules 4 to 8) is the change of state variables and specifies which node to be visited next.

In the first two steps of the algorithm 5.1, we choose the next node to be visited, in Depth First Search manner and the currently active object. Rules 3.1 to 3.5, 4, 5 and 6 of the operation semantics choose the appropriate transition and find the next active node (in_n.nID).

All nodes (activity node, initial node, decision node and join node) have single outgoing transitions, except for fork node and the next active node will be the target node of the outgoing transition. For decision node, even though there is more than one possible outgoing transitions only one of them will be taken based on the guard condition. The same is achieved in the algorithm 5.1 too. That is, except for the fork node, for all other nodes, the next node is taken using the common step (i.e, step 1).

In step 3 of algorithm 5.1, activity nodes are processed. If OCL statements are included in the node, that will also be converted to the source code. As per the operational semantics Rule 5, $(in\_n.\psi -> execute(n.\psi))$, if there is any OCL statement in the node, it has to be executed before finding the next node to be visited. The Boolean variable to activate the next node in_n.nID has to be activated

$$(in\_n.nID = next(in\_n.nID) \left| \vee_{t \in n.in \cup n.out, t\_taken defined} t\_taken \right|$$
$$\vee_{t \in n.out.tgt.out, t.src \in CN_{fork}} t\_taken \Big| \vee_{t = n.out.tgt.out, t.src \in CN_{join}} t\_taken \Big)$$

The assertion for this can be mentioned as

P1→ Execute OCL statement in the node (if any)

P2→ Activate the next node to be visited, target node

As per the operational semantics Rule 3.1, the transitions from the activity node is handled as follows

$$t\_taken := in\_t.src.nID \,\&\, \vee_{t.src \in AN} in\_t.src.\psi \,\&\, \vee_{t.tgt \in AN} next(in\_t.src.\psi) \,\&$$
$$\vee_{t.src \in AN} in\_t.src.e \,\&\, \vee_{t.tgt \in AN} next(in\_t.src.e) \,\&$$
$$\text{next(in\_t.tgt.nId)} \,\&\, next(activeNode = t.tgt.nID);$$

The transition is taken if the source node is visited and the OCL statements and external events have been processed (if any). Next target node is set as the activeNode. From the above semantics we can formulate the assertion as follows.

P3→ Traverse to next node if all OCL statements are executed and events are handled.

These three assertions are satisfied in the algorithm steps 1 and 3, as shown below.

```
   Perform Depth First Search on $AG_{ocl}$ by keeping *start* as starting vertex
1.visit next node N.out_edge.target
2.identify currently active object, ca_obj,
3.if(N.type = = activity)
        3.1 create statement ca_obj.actionName();
                write to main() if currentThread='main'
                otherwise write to run() method where threadname=currentThread
        3.2  If this node element contains OCL element <ocl>
                3.2.1 Call methodDefinition(P,ocl)
```

P1 is attained in step 3.2, in which the OCL statements have been processed and generate the corresponding source code. Algorithm 5.2 is used for converting OCL statements to source code.

P2 is attained in step 1, since it finds out the next node to be visited using the outgoing edges from the current node and in a depth first search fashion.

P3 is attained in steps 3.1 and 3.2. Step 3.1 handles the activities and events in the node. The events are handled as the actions in the activity node and the corresponding event handling functions will be called. The implementation of the activities and the event handling are done in a similar way. So the events are also handled in step 3.1. In step 3.2, the OCL statements have been processed. The transition to next node is preceded by these two steps and hence, assertion P3 is ensured.

In step 4 of algorithm 5.1, the decision node is processed. As per the operational semantics, an outgoing edge with true guard condition will be taken. The next node to be visited is set to the target node of the outgoing edge.

$$\begin{aligned}&\forall t \in E \wedge t.src \in CN_{decision} : t\_taken := in\_t.src.nID \,\& \\ &\qquad t.G \,\& \,\text{next(in\_t.tgt.nId)} \,\& \\ &\qquad next\big(activeNode = t.tgt.nID\big);\end{aligned}$$

The assertion for this can be mentioned as

P4→ Traverse to the next node if the guard condition is true and the next active node has been set as the target node.

This assertion is satisfied in the step4 of the algorithm 5.1, as shown below.

```
4.  else if(N.type = = decision)
      4.1 For each N.out_edge, create "if(N.out_edge guard)" statement
            write to main() if currentThread='main'
            otherwise write to run() method where threadname=currentThread
```

Step 4 takes each decision node and step 4.1 takes each outgoing edge from the decision node. The guard conditions for each edge is made a part of the code generated, so that the right path will be taken based on the true guard condition. After processing all the edges, step 1 is executed to find the next node using DFS. So we can say,

> **P4** is attained in step 4 and step 1 since the source code is generated for each edge along with its guard condition and the next active node is chosen based on the guard condition.

In step 5 of algorithm 5.1, the fork node is handled. As per the operational semantics, when the currently active node is a fork node, all control variables of the fork node (*in_Ft.tgt.nId*) has to be activated. $\forall fn \in CN_{fork} : (activeNode = fn.nodeID -> \wedge_{t \in fn.out}\, next(in\_Ft.tgt.nId))$.

Change the control variables of the fork node (*in_Ft.tgt.nId*) whenever an incoming or outgoing edge to the fork node is taken.

$$\forall fn \in CN_{fork} \forall t \in fn.out : \left(in\_Ft.tgt.nId = next(in\_Ft.tgt.nId) \middle| \vee_{t' \in fn.in \cup t} t'\_taken \right)$$

The outgoing edge from a fork node has to be taken if its control variable (*in_Ft.tgt.nId*) is activated and the next active node is set to the target node of the edge.

$$\forall t \in E \wedge t.src \in CN_{fork} : t\_taken := in\_Ft.tgt.nId \,\&\, \text{next(in\_t.tgt.nId)} \,\&$$
$$next(activeNode = t.tgt.nID);$$

So the assertions can be formulated as follows.

> **P5→**if the current node is a fork node activate the control flow variables of the outgoing edges
>
> **P6→** change the value of control variable whenever an in or out edge has been taken.
>
> **P7→** an out edge is taken when the control variable is enabled and next active node is set as the target node.

These three assertions are satisfied in the steps 1 and 5 of algorithm 5.1, as shown below.

```
5. else if($N$.type = = fork)
    5.1.    count the number of child nodes, say c.
    5.2.    while ( c>=1 )
        5.2.1.    create and start thread with name parentThread _$t_c$ ,
                  write to main() if currentThread='main'
                  otherwise write to run() method where threadname=currentThread
        5.2.2.    push parentThread_$t_c$ to stack
        5.2.3.    decrement c by 1
    5.3. if (currentThread = 'main')
        5.3.1. Call CodeGeneration($AG_{ocl}$,P, currentThread, currentThread,
               currentForkNode)
        5.3.2. Pop sub threads from stack and call CodeGeneration($AG_{ocl}$,P,
               currentThread,parentThread_$t_c$, currentForkNode) till stack is empty
    5.4.Else if(currentThread $\neq$ 'main')
        5.4.1. Call CodeGeneration($AG_{ocl}$,P, currentThread, currentThread,
               currentForkNode)
        5.4.2. modify run() method in Thread class
        5.4.3. add "if(threadname==parentThread_$t_c$){ " statement to run() method
        5.4.4. call
               CodeGeneration($AG_{ocl}$,P,currentThread,parentThread_$t_c$,currentForkNode)
```

Step 5.1 counts the number of outgoing edges (or child nodes) of the fork node. Step 5.2 takes each edge (path) separately and keep ready for processing. Steps 5.3 and 5.4 create source code for each parallel path between the fork and join nodes. The next node to be visited is taken using recursion and DFS. So we can say,

P5 and **P6** are attained in step 5.1 and step 5.2. All outgoing edges from the fork nodes are processed in these steps.

P7 is attained in steps 5.3 and 5.4. All the outgoing paths from the fork node have been visited using recursion. Threads are used for implementing the control flows of parallel paths.

In step 6 of algorithm 5.1, join node is handled. As per the operational semantics, if the currently active node is a join node, activate its incoming edge, or the last traversed edge (*in_Jn.src.nId* set to true).

$$\forall jn \in CN_{join} \forall t \in jn.in: \quad \wedge(activeNode = jn.nodeID -> next\left(in\_Jt.src.nId\right));$$

The control variable of the join node (*in_Jn.src.nId*) is changed whenever an incoming or outgoing edge is taken.

$$\forall jn \in CN_{join} \forall t \in jn.in: \qquad \left(in\_Jt.src.nId = next\left(in\_Jt.src.nId\right) \middle| \vee_{t' \in t \cup jn.out} t'\_taken\right);$$

The outgoing edge of a join node is taken only when all the incoming edges are taken (activated) and the next active node is set to its outgoing edges's target node.

$$\forall t \in E \wedge t.src \in CN_{join} : t\_taken := \wedge_{t \in t.src.in}\, in\_Jt.src.nId \,\&\, \mathrm{next(in\_t.tgt.nId)} \,\&\, next\left(activeNode = t.tgt.nID\right);$$

The assertions can be formulated as follows.

P8→if the current node is a join node activate the control flow variable of the incoming edge. Repeat it until all incoming edges get activated.

P9→ change the value of control variable whenever an in or out edge have been taken.

P10→ an out edge is taken when all the control variables are enabled and the next active node is set as the target node.

These assertions are satisfied in steps 1 and 6 of algorithm 5.1, as shown below.

```
6.  else if (N.type = = join)
    6.1.  if(currentThread=='main')
          return
    6.2.  else if(currentThread==parentThread)
          return
    6.3.  else if(parentThread=='main')
          add "currentThread.join()" to main() method
          return
    6.4.  else if currentThread ≠ parentThread
          add "currentThread.join()" to run() method where threadname=parentThread
          return
```

Steps 6.1 and 6.2 look for the next parallel path (sibling) without processing the join node. If the current thread is a child thread it has to wait till all other threads reach join node. it is done in steps 6.3 and 6.4. Transition from the join node is taken only when all the parallel paths have been reached the join node. it is ensured with the recursive calls in steps 5.3 and 5.4. Hence,

P8and **P9** are attained in steps 6.1 to 6.4. Waiting for all edges to be taken is implemented using the join() function. Main thread will wait till all the child threads complete their execution.

P10 is attained using step 1. The recursive calls made at the fork node will be returned in steps 6.1 to 6.4. After returning all threads,(ie, all incoming edges to the join node are traversed) the control goes to step 1 and traverse to the next node after the join node.

In step 7 of algorithm 5.1, the merge node is handled. As per the operational semantics, the merge node is a control node that brings together multiple alternate flows. The outgoing edge from a merge node is taken when its previous node is visited and the next active node is set.

$$\forall t \in E \wedge t.tgt \in CN_{merge} : t\_taken := in\_t.src.nID \,\&\, \text{next(in\_t.tgt.nId)} \,\& \\ next\left(activeNode = t.tgt.nID\right);$$

The assertion for the same is written as follows.

P11→ if the previous node has been visited activate the target node and set the next active node to the target node.

Step 7 of the algorithm 5.1 satisfies this assertion, as shown below.. When a merge node is encountered, the control goes to step 1 and fetch the next node to be visited.

```
7.  else if (N.type = = merge)
      go to step 1
```

So we can say that,

P11 is achieved in step 7 since it redirects the execution to step 1, finding next node to be visited.

In step 8 the final node of the activity diagram is processed. As per the semantics, when all nodes have been visited, the next active node is set as ‘no operation’.

$$\forall n \in CN_{final} : \wedge in\_n.nID -> next(activeNode = nop) \, \& \\ \left(\vee in\_n.nID \middle| \vee_{t \in T, t\_taken defined} t\_taken \right);$$

So the assertion can be written as follows.

P12→ if all nodes have been enabled then set next active node to *‘no operation’.*

This assertion is attained in step 8 of algorithm 5.1, as shown below.

```
8. else if(N.type = = finalNode)
   return P
```

When the final node is encountered the algorithm stops processing the activity diagram and returns the source code generated. No further processing is done. The steps 1 to 7 ensure that all nodes have been visited before reaching the final node. Hence it is true that,

P12 is attained in step 8 since it stops processing the AG and return the code generated.

The algorithm is capable to handle and generate implementation code for activity nodes with OCL, decision nodes, fork nodes, join nodes, merge nodes, initial and final nodes. The algorithm 5.1 accepts any activity diagram with activity node, decision node, fork node, join node, merge node, initial and final nodes and convert it to source code. Hence, the correctness of the algorithm is proved.

Algorithm 5.2 is used to convert the OCL statements to corresponding source code. It processes five types of the OCL statements, like initial value, object, pre condition, Post condition and operation body. How each type of the OCL statement is converted to source code is mentioned in steps 1 to 5. The initial value of an attribute will be assigned in step 1. An instance of a class of the given name is generated in step 2. The pre condition and post condition statements are generated in steps 3 and 4. In step 5 the method body for the specified method is generated. So it is evident that the algorithm 5.2 can handle any kind of OCL statements present in the activity diagram and can generate corresponding implementation code. Hence, the correctness of the algorithm is proved.

In order to demonstrate the working of these algorithms we present a case study in the next section. The money withdrawal operation of a bank ATM is taken as the case study.

5.6 Case Study

In this section we consider the process flow of the money withdrawal operation in a bank ATM. We have considered a subset of the operation in the ATM for our case study. We consider that the process flow starts with secret code verification as shown in Figure 5.18. If the code is correct, the machine will proceed with transaction and ask for amount to withdraw.

If the secret code is incorrect, the machine displays error message and also rechecks the code. If the code found correct, the machine will ask for amount. Otherwise, the ATM rejects the transaction.

After reading the amount for withdrawal, there are two concurrent processes, *dispense cash* and *prepare the receipt*. Finally the transaction will be closed and the receipt will be printed out. The guard conditions associated with the edges for the decision making nodes are given in the model. The generated code can be in the following format.

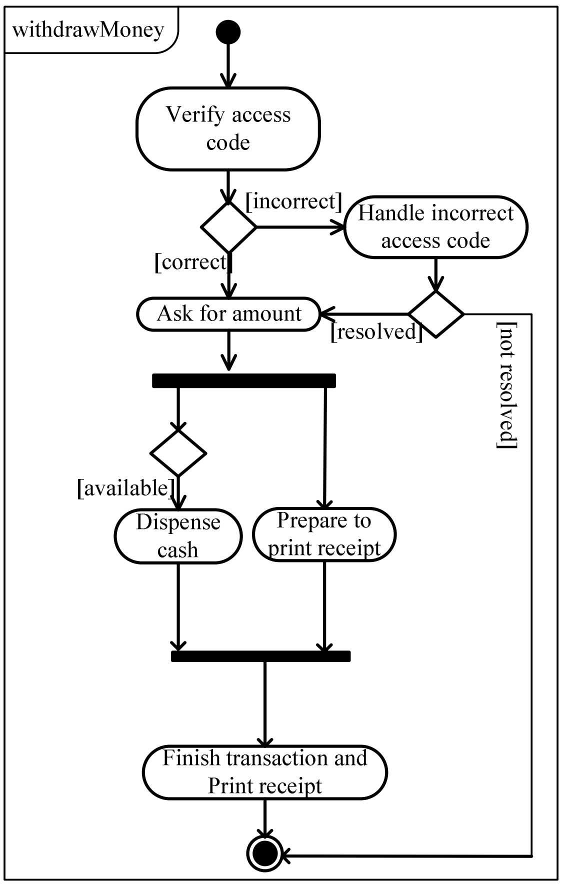


Figure 5.18: Activity diagram for money withdrawal from ATM machine (without OCL)

```
/******************** Main Class *************************/
public class WithdrawMoneyMainClass
{       .......
        .......
        public static void main(String arg[])
        {   atm_obj_01.verifyAccessCode();
            if(correct)
           {     atm_obj_01.askForAmount();
                 NewThread thrd_01=new NewThred("thrd_id_01");
                 if(available)
                 {        atm_obj_01.dispenseCash();
                          thrd_01.join();
                          atm_obj_01.finishTransaction();
                 }
            }
            if(incorrect)
            {     atm _obj_01.handleIncorrectAccessCode();
                  …………………
                  …………………
            }
        …………..
        }
}
/************** Class for Managing Fork & Join ***************/
public class NewThread implements Runnable
{       …………………
        …………………
        public void run()
        {       if(name.compareTo("thrd_id_01")==0)
                {       atm_obj_01.prepareReceipt();        }
        }
}
/********************* Context Class *********************/
public class WithdrawMoneyClass
{       public void verifyAccessCode( ){ /* TODO CODE HERE*/ }
        public void askForAmount( ){ /* TODO CODE HERE*/ }
        public void dispenseCash ( ){ /* TODO CODE HERE*/ }
        public void finishTransaction ( ){ /* TODO CODE HERE*/ }
        public void handleIncorrectAccessCode( ){ /* TODO CODE HERE*/ }
        public void prepareReceipt( ){ /* TODO CODE HERE*/ }
}
/******************* CODE ENDS ************************/
```

Some details are still missing. Using OCL, we try to include some more details that are useful for the system implementation as shown in Figure 5.19. For example, 'Handle incorrect access code' is a confusing term. How to handle the incorrect code is not derivable from the activity name. So, we add the OCL statements, as operation body, to add these details.

```
<<operation body>>
context withdrawMoney :: handleIncorrectAccessCode()
body:  self. displayIncMsg()
       self.recheckCode()
```

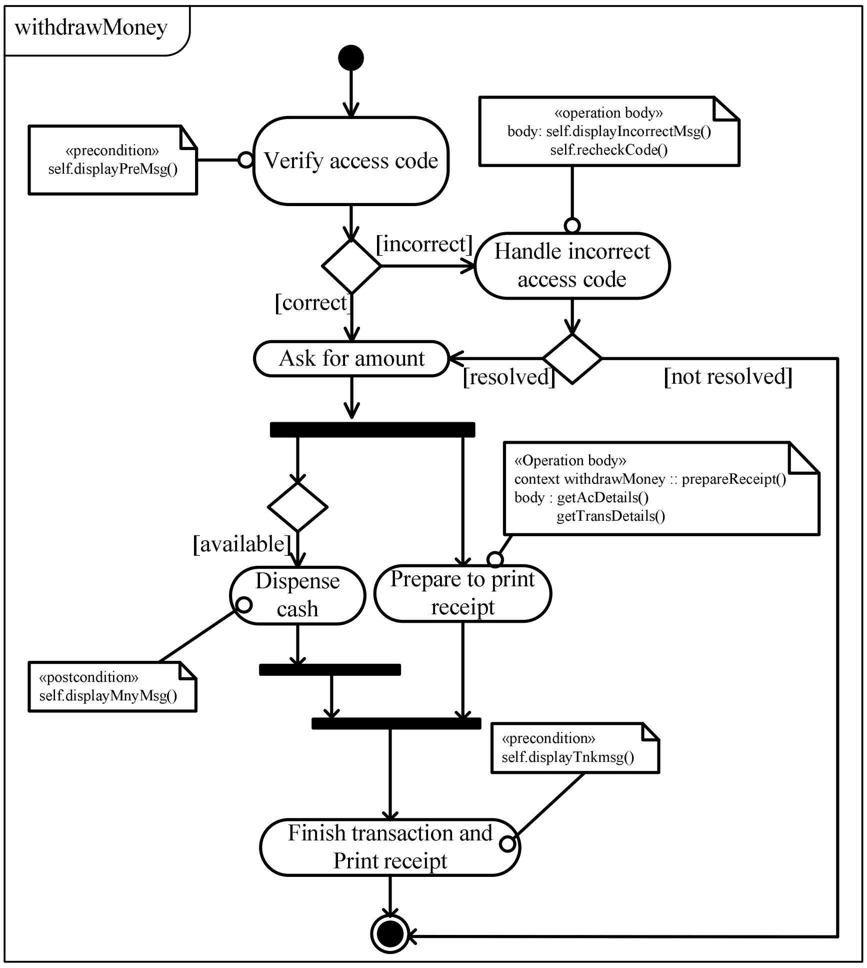

withdrawMoney
«precondition» self.displayPreMsg()
Verify access code
«operation body» body: self.displayIncorrectMsg() self.recheckCode()
[incorrect]
Handle incorrect access code
[correct]
Ask for amount
[resolved]
[not resolved]
«Operation body» context withdrawMoney :: prepareReceipt() body : getAcDetails() getTransDetails()
[available]
Dispense cash
Prepare to print receipt
«postcondition» self.displayMnyMsg()
«precondition» self.displayTnkmsg()
Finish transaction and Print receipt

Figure 5.19: Activity diagram for money withdrawal from ATM machine (with OCL)

Similarly, the activity '*prepare to print receipt'* is also ambiguous. The details of the preparations are given as the operation body in OCL statements.

<<operation body>>

context withdrawMoney::prepareReceipt()

body: getAcDetails()

getTransDetails()

Moreover, some pre and post conditions are given along with the activities '*verify access code'*, *'finish transaction and print receipt'* and *'dispense cash'* respectively.

<<precondition>>

context withdrawMoney:: verifyAccessCode ()

self.displayPrcMsg()

<<precondition>>

context withdrawMoney:: finishTransaction ()

self.displayTnkMsg()

<<postcondition>>

context withdrawMoney:: dispenseCash ()

self.displayMnyMsg()

These OCL expressions help to improve the code generation, especially the generation of the method definitions. The OCL statements given in Figure 5.19 include pre- and post conditions and operation body. So these statements help us to modify the context class WithdrawMoneyClass. The code generated from the OCL enhanced activity diagram is as follows.

```
/******************** Context Class ********************/
public class WithdrawMoneyClass
{       public void verifyAccessCode( ){
                /* TODO CODE HERE*/
                //-----precondition-----
                displayPreMsg();
                //------------------------
        }
        public void askForAmount( ){ /* TODO CODE HERE*/ }
        public void dispenseCash ( ){
                /* TODO CODE HERE*/

                //-----postcondition-----
                displayMnyMsg();
                //------------------------
        }
        public void finishTransaction ( ){
                 /* TODO CODE HERE*/
                //-----precondition-----
                displayTnkMsg();
                //------------------------
        }
        public void handleIncorrectAccessCode( ){
                /* TODO CODE HERE*/

                //-----operation body-----
                displayIncMsg();
                recheckCode();
                //------------------------
        }
        public void prepareReceipt( ){
                /* TODO CODE HERE*/

                //-----operation body-----
                getAcDetails();
                getTransDetails();
                //------------------------
        }
}
/****************** CODE ENDS ***********************/
```

We will get promising results when we apply our method with low level activity diagrams, for example, describing the logic of an operation using activity diagram. High level activity diagrams outline the business process or task and its flow. So, the realization of those diagrams will end up in a series of method calls, as given in the above example.

5.7 Implementation of Automatic Code Generator

The code generation process and the algorithm for code generation are implemented in Java and we developed a tool called ActivityOCLKode. It has mainly three modules; ActivityOCLModeler, OCL checker and ActivityOCL Code generator as shown in Figure 5.20.

The ActivityOCL Modeler, as the name indicates, helps the user to model the system design using activity diagram as well as OCL statements. The modeler supports all the essential elements in UML activity diagram. It supports the activity node, decision node, fork node and join node. Moreover, the modeler gives us options to include OCL statements with each element in the activity diagram. In the current version, ActivityOCLKode supports OCL expression for pre- and post conditions, operation body and guard conditions.

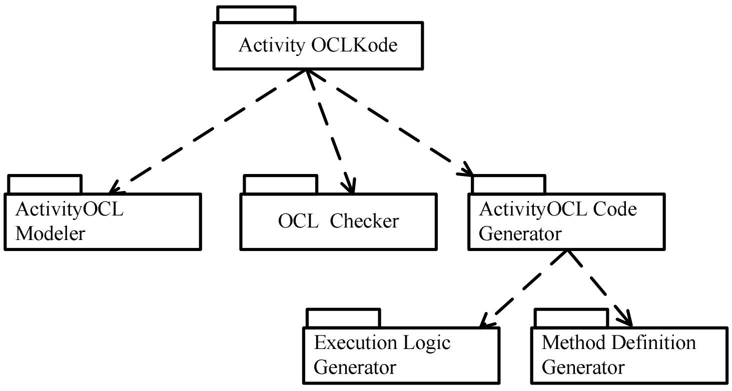

Activity OCLKode
ActivityOCL Modeler
OCL Checker
ActivityOCL Code Generator
Execution Logic Generator
Method Definition Generator

Figure 5.20: ActivityOCLKode Architecture

The modeler has some additional features to save the activity diagram in JPEG format. Internally, the modeler saves the model as XML document. It also provides XML parser to parse the XML document and to get the object tree of the document. This is necessary to retrieve data from XML documents. In addition to parser, the modeler provides a tree view option which gives the tree structure of the XML document.

Another important part of the ActivityOCLKode is the OCL checker. The user designs the process flow, or the system design using activity diagram and the finer details will be furnished using OCL expressions. These expressions can be made hidden in the diagram or it can be explicitly visible. Since OCL is a formal language, it follows some syntax rules [124]. So, before compilation we are supposed to check the well formedness of the expressions. This is the duty of the OCL Checker. It checks for syntax errors in the OCL statements. The parsed XML document is used for this purpose. After parsing the XML document we get an in-memory tree representation of the XML document. Extract all the nodes with name '*<OCL>*' from the object tree and do the checking. We check the syntax of the pre- and post conditions, operation body and guard conditions. The type of the expression is mentioned along with the <OCL> tag. If there is any error, it will be reported to the user using appropriate error messages. Then the user can make necessary changes to the OCL statements and regenerates the XML document. This will continue till the model is error free.

Finally, the Activity OCL code generator is the heart of the ActivityOCLKode. We use Model Driven Development approach for code generation. The tool helps to model process flow using activity diagram and then it will be converted to XML. The tool uses Java and XML for platform specific modeling (PSM).

The tool takes the XML document that is built after checking and correcting the OCL statements. The OCL statements will not be processed separately after regenerating XML. For code generation, we follow the algorithms which are mentioned in the previous section. The code generator has two main components; execution logic generator and the method definition generator. The first one uses algorithm 5.1 for code generation and the second one uses algorithm 5.2 for method definition generation.

As per the algorithm, the overall execution logic is formed by checking each node in the activity diagram starting from the initial node. The edges give the flow of execution. Each activity is converted as a function call. These functions can be defined with the help of OCL. The decision making nodes are converted to if…else statements with the guard conditions of the associated edges.

The fork and join nodes are implemented as start of threads and join of threads. Concurrent actions are given between the same fork and join nodes. Threads are used for concurrent actions in Java. Independent threads will be initiated for each path from the fork node. One of them may be the main thread. There is a unique identifier for each thread as t_forkid_no. the presence of fork id in each thread id ensures its uniqueness.

Implementation of method definition is supported by OCL expressions. If we add more OCL statements to specify the operation body and other details, the generated code will be better. The use of OCL increases the percentage of code generated. Each method definition is packed in the order; pre condition, operation body and post condition. If these elements are given in the activity diagram using OCL, it will contribute much to the code generation.

5.8 Evaluation

The proposed method is implemented and evaluated. The evaluation is done in three dimensions; the type of code generated, percentage of code generated and the time complexity for code generation.

5.8.1 Type of code generated

The implementation of a system design normally contains the class definitions, method declarations, method definitions, method calls, constructors, method & variable declarations and variable initialization. We evaluated our approach in this regard. It is summarized in the Table 5.1.

The ActivityOCLKode can successfully generate the definition of the context class with method declarations, the supporting class for implementing fork and join and the main class and main method for the application using logical method calls.

Variable initializations and method definitions can be generated with the help of OCL statements in the activity model. The skeletal code of the class constructors will also be generated by ActivityOCLKode. The local variable declarations and its manipulations cannot be generated automatically by ActivityOCLKode.

Table 5.1: Type of code generated by ActivityOCLKode

Sl. No	Type of code	Generated by ActivityOCLKode
1	Variable declaration	No
2	Method declarations	Yes
3	Method calls	Yes
4	Class definitions	Yes
5	Main class and Main method	Yes
6	Variable initialization	generated with the help of OCL statements
7	Method definitions	generated with the help of OCL statements
8	Constructors	Skeletal code will be generated

5.8.2 Percentage of code generated

The ActivityOCLKode is tested for its performance. We have tested the tool with system designs having different complexity levels. Process flow designs with decision making nodes, concurrent flows etc have been taken for evaluation. For a systematic evaluation we have taken activity diagrams with different levels of details like 1, 2, 3 and 4. In level 1, we represent the system model with simple activity diagram which contains only activity nodes. Level 2 is the activity model of the system which includes decision nodes too. Similarly, level 3 includes concurrent nodes. Moreover, we compared the code generated from activity diagrams with and without OCL expressions.

The main logic of the method definitions are included using OCL. The pre- and post conditions are added with operations give a precise boundary for the implementation. All these things improved the code generation from activity diagrams.

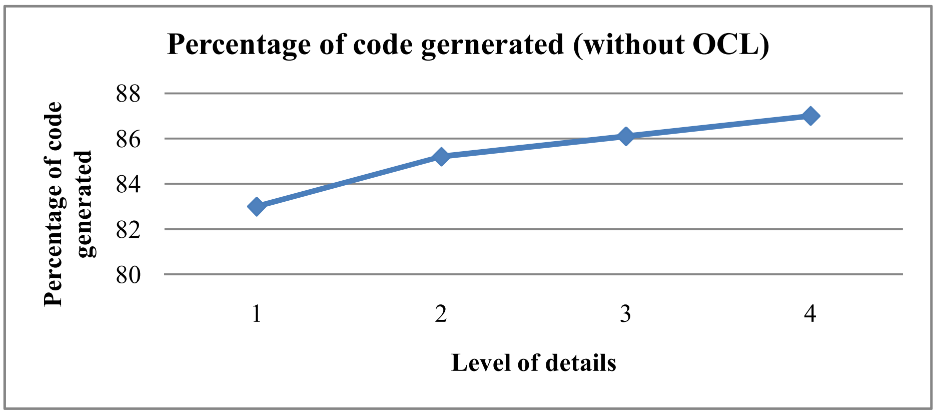


Figure 5.21: Percentage of Code Generated without OCL

We evaluated the code generated from activity diagrams which do not contain OCL expressions. It gives more than 80% code coverage. The activity diagrams without decision or fork nodes are tested for code generation and they give 83% of the source code. Similarly, we evaluated the percentage of code generated from activity diagrams having decision making nodes. It evaluates to 85.2%. The activity diagrams with concurrent paths give 86.1% code coverage.

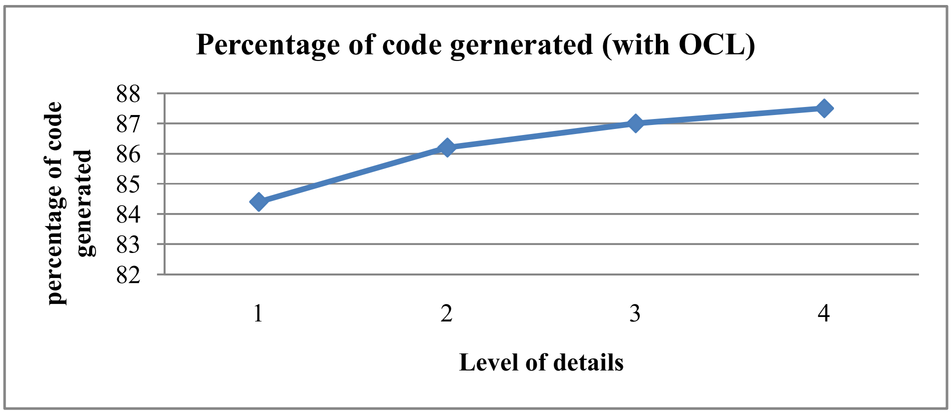


Figure 5.22: Percentage of Code Generated with OCL

Finally the complex diagrams with both decision and fork nodes give 87% of code. As a next step, we evaluated the code generated from the activity diagrams which are supported by OCL. We could see that the inclusion of

OCL improved the code generation. Simple diagrams give 84.4%, diagrams with decision making give 86.2, diagrams with fork nodes give 87% and diagrams with decision and fork nodes give 87.5%. Figure 5.21 and Figure 5.22 show the graphical representation of the performance evaluation.

5.8.3 Time complexity for code generation

We have evaluated the performance of ActivityOCLKode in terms of the time complexity too. For this we considered activity diagrams with different complexity levels. An activity diagram may contain activity nodes, decision nodes, merge nodes and concurrent activities with fork and join.

If the activity diagram includes only the activity nodes, then it can be called as simple diagram falling in complexity level 1. If the diagram contains decision node and merge node, the complexity increases slightly; still we keep it in level 1. If there is a fork & join nodes in the diagram, it increases the complexity, since it introduces concurrent activities. We keep this kind of diagrams in level 2. Now, we can have multiple decision nodes in a diagram, it lies in level 3. Similarly we can have different combinations of decision nodes and fork & join nodes to get the complexity levels from 4 to 9. The introduction of each fork node to the diagram increases its complexity, since each fork initiates one or more threads of execution.

We executed ActivityOCLKode for different types of activity diagrams which falling under any one of the above nine complexity levels. The complexity levels and the time taken for execution are shown in Table 5.2. Complexity levels 1 to 4 includes the decision making nodes and two or three concurrent execution paths of the activities. Levels 5 to 9 include complex activity models which contain more than one fork & join nodes with two or more parallel paths.

Table 5.2: Levels of complexity and the execution time in milliseconds

Complexity Level	Type of nodes included	Exec time in ms
Level 1	Decision nodes and activities	6.94
Level 2	Concurrent activities	9.52
Level 3	Multiple decision nodes	10.84
Level 4	Decision with concurrent activities	11.35
Level 5	Multiple fork & join	11.58
Level 6	Multiple decision nodes with concurrent activities	12.35
Level 7	Multiple fork & join with more parallel activities	12.60
Level 8	Multiple decision, fork & join, and merge nodes	13.38
Level 9	More than 5 fork & join	14.59

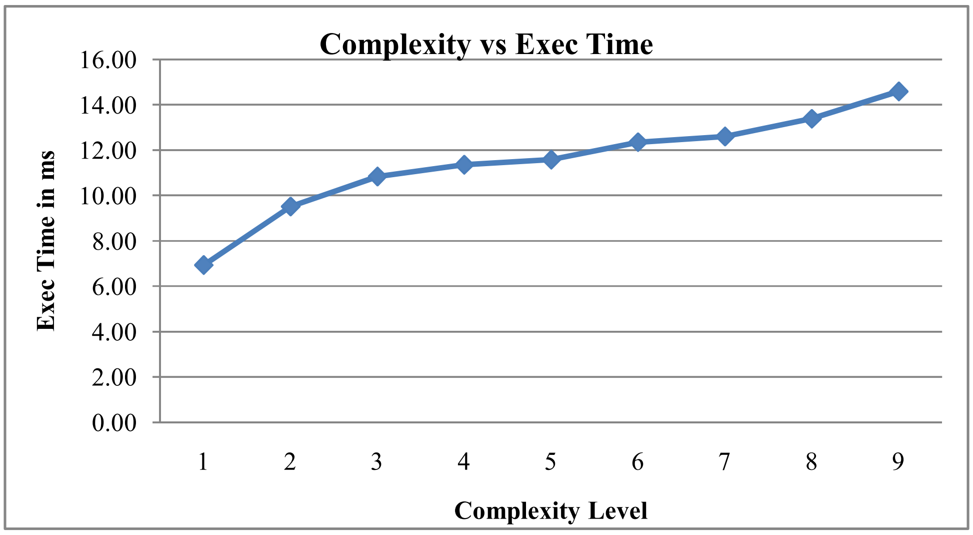


Figure 5.23: Complexity Vs execution time

An activity model in complexity level 1 will be converted to source code in 6.94 milliseconds. The execution time for Level 2 is 9.52 ms, Level 3 is 10.84 ms, Level 4 is 11.35ms, Level 5 is 11.58 ms, Level 6 is 12.35ms, Level 7 is 12.6 ms, Level 8 is 13.38 ms and Level 9 is 14.59ms. The analysis shows that the execution time slightly increases with the increase in the number of concurrent paths in the activity model. Whenever the ActivityOCLKode finds a fork node in the diagram, it will create (n-1) threads, where n is the number of child nodes of the fork node. Tracing out each parallel path and update the source code accordingly steals some execution time. So, the execution time increases with the number of fork nodes. The analysis is shown in Figure 5.23.

5.9 Conclusion

This chapter presents the semantics for enhancing UML Activity diagram with OCL, in the form of meta models and operational semantics. The literatures related to the code generation from the UML Activity diagrams lack a formal semantic for enhancing the activity models with OCL.

We have developed several metamodels for enhancing activity diagram with OCL and presented in this chapter. This gives a clear picture about how to incorporate OCL in UML activity diagram. The operation semantics proposed in this chapter explain how an OCL enhanced activity diagram works. The proposed algorithms give a proper guideline for the code generation from OCL enhanced activity diagram. To the best of our knowledge no other research outcome reported a precise algorithm for code generation from OCL enhanced UML activity diagram.

ActivityOCLKode, the tool implemented based on proposed algorithm, provides a user friendly environment for the users to model the process flow based software systems. The evaluation of the tool shows the proposed method of code generation helps us to generate more than 83% code. When we add OCL with the activity diagrams, this raises up to 84.4%. The code, generated from OCL, is very crucial since it includes method definitions and the specific pre- and post conditions. Moreover, the time required for code generation based on the proposed method is 11.46 milli seconds (average) only.

The proposed code generation approach reduces the software implementation and documentation efforts. The use of OCL improves the percentage of code generated. The use of XML to save the models in text format improves the portability of the models. UML models, OCL and XML all are widely accepted and used in software industry and so the proposed method can be easily adapted to the software development process in the software industry.

Chapter 6 CODE GENERATION FROM ACTIVITY MODELS ENHANCED WITH INTERACTIONS

6.1 Introduction

Activity diagram can show the group of activities done by different objects in the system. We can specify which object is responsible for which activity. This is a unique feature of activity diagram compared to other behaviour diagrams like state chart diagrams. This feature, which is not exploited so far, is very much useful in automatic code generation.

In this chapter, we define a method to fine tune the activity diagrams for improved code generation. We tried to associate activity diagram with sequence diagram to include object interactions in activity diagram. We introduced a formal method for this association. Even though activity and sequence diagrams are behavioural models, they model the system from different perspective. First one is activity centric and the latter is object interaction oriented. If we put them together we can generate more complete source code because, activity diagram contributes to control flow and class definitions and sequence diagram contributes to method definitions and object interactions.

The concepts like activity node, work flow, decision node are easy to implement, since there are built in programming constructs like method invocation statements, if...else statements, etc. When we consider more advanced and expressive features like fork & join nodes, the implementation is not so straight forward. We surveyed many research publications to find a method to handle concurrency (fork & join) in the activity diagrams. Unfortunately, we could not find any method/algorithm to do so. Hence, we formulated an algorithm to convert concurrent activities in the activity diagram to source code. In addition, we introduced algorithms for code generation from activity and sequence combined models. Our method gives a well defined algorithm for code generation which is lacking in other published works in this field of research.

In this chapter, we use UML 2.0 activity & sequence diagrams for system design. We have evaluated our method by developing a tool called AutoKode.

The main contributions of this chapter are as follows:

- It provides formalization for associating activity diagram with sequence diagram.
- It provides a method to generate control flow from activity diagram and method definition and object interactions from sequence diagrams.
- It provides a better algorithm for code generation from activity and sequence diagrams.

6.2 Associating Activity Diagram With Sequence Diagram

Associating activity diagram with sequence diagram will help us in code generation from activity models. In this section, we explain a method to associate activity and sequence diagrams for workflow modeling. The basic idea behind this is that all the control flow and data flows are depicted using the activity diagram and the method invocations (communication between objects) are depicted using sequence diagrams. Each activity in an activity diagram can be expanded using a sequence diagram. This will help us to include more details to the activity diagram. That is we have one activity diagram and many sequence diagrams associated with it as shown in Figure 6.1. Each activity node in the activity diagram generates a method declaration and the corresponding sequence diagram generates the method definition as a sequence of method calls. The execution control flow will be based on the control flow in activity diagram.

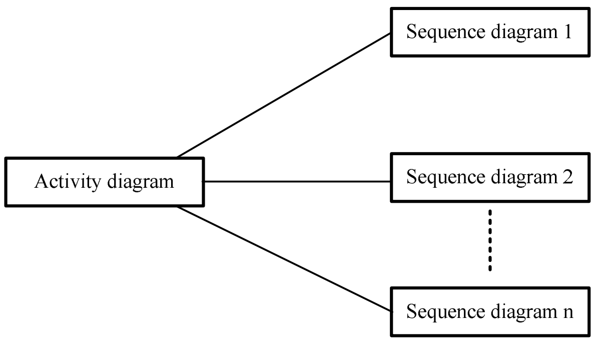


Figure 6.1: Associating activity diagram with sequence diagrams

Formal definition for the association of activity and sequence diagrams is presented in section 2.1 and section 2.2 explains the mapping of the definition elements to the UML 2.0 Activity diagram meta model.

6.2.1 Formal definition for associating activity diagram with sequence diagram

In this section, we formally define activity diagram and sequence diagram. The definition reveals the association between them. This formal definition shows the elements in activity diagram and sequence diagram that are considered for work flow modeling and collaboration modeling respectively. This formal definition is used in the algorithm for prototype generation, which is explained in the next section. In the earlier versions (UML 1.x), activity diagram had been considered as an extension of the state machine. As per UML2.0, it is considered as a graph [139, 103]. So we use the term *Activity Graph* to represent activity diagram in our definition.

Formalization of Sequence Diagram: A sequence diagram, $\mathcal{SD}$, is a tuple. It is a set of objects σ and the interactions m between them. The interactions are through message passing. As Li [76] defines message, it has four main components – action, sender ob_i, receiver ob_j and sequence order of the message. The action can be of five types; synchronous message, asynchronous message, return, create and destroy.

$$
\begin{aligned}
\mathcal{SD} &= (\ \sigma,\ m\ )\,where\\
&\quad \sigma = \{x \mid x\ is\ an\ object/actor\}\\
&\quad m = \{msg \mid msg\ is\ a\ message\}\\
&\qquad where\ msg\ is\ a\ tuple.\\
&\quad msg = \left(\ ob_i : C_i,\ ob_j : C_j,\ action,\ order\right)
\end{aligned}
$$

[σ– objects, m– messages, ob_i – source object of class C_i, ob_j – target object of class C_j, *action* – method call, *order*- the sequence number of the message in the current sequence diagram, C_i and C_j are classes].

Formalization of Activity Graph **:** An Activity Graph, $\mathcal{AG}_{sd}$, is a hextuple which contains nodes $\mathcal{N}$, edges $\mathcal{E}$, events e, guard conditions $\mathcal{G}$, local variables var and set of objects σ. A node can be of two types, *ActionNode* $\mathcal{AN}_{\alpha}$ and

ControlNode CN. AN_{d} includes *action node, acceptEvent node, sendSignal node* and *CallBehaviorAction* (CBA) node. The *CallBehaviorAction node* is used to represent a function call in the activity diagram. It is implemented as a call to a sequence diagram which details the function. CN includes *initial node*, *actionFinal node*, *flowFinal node*, *decision node, merge node and fork and join node*.

$$
\begin{aligned}
AG_{sd} &= (N,\ E,\ e,\ G,\ var, o),\ \textit{where} \\
N &= AN_{d} \cup CN \\
AN_{d} &= \{\textit{action},\ \textit{accpetEvent},\ \textit{sendSignal}\} \cup CBA \\
CN &= \left\{ \begin{matrix} \textit{initialNode},\ \textit{actionFinal},\ \textit{flowFinal}, \\ \textit{decision},\ \textit{merge},\ \textit{fork},\ \textit{join} \end{matrix} \right\} \\
CBA &= \{x \mid x \textit{ is an} SD \textit{ or } AG\} \\
E &= \{\ (x,y) \mid \textit{transition from } x \textit{ to } y\} \textit{ where } x,y \in N \\
e &= \{x \mid x \textit{ is an external event}\} \\
G &= \{x \mid x \textit{ is a guard expression}\} \\
var &= \{x \mid x \textit{ is a local variables}\} \\
o &= \{x \mid x \textit{ is an object / actor}\}
\end{aligned}
$$

The Activity Graph, AG_{sd}, contains *callBehaviorAction* nodes (CBA), which are associated with sequence diagram, since CBA is implemented using SD. In both AG_{sd} and SD, set of objects/actors o is common.

6.2.2 Mapping of the definition elements to UML meta model

The abstract syntax of AG_{sd} is shown in Figure 6.2. The Activity diagram (AG_{sd}) may contain many activities. The element Activity Diagram in the figure represents the AG_{sd}. The mapping is shown in Table 6.1. AG_{sd} contains many activities. Each *Activity* is associated with an *Object* which is responsible for the activities to be done. There will be associated *Events* and variables. Each activity is composed of *Activity Edges* and activity nodes. Each activity edge can be either *Control flow* or *Object flow*. Each *Activity*

Node can be *Control* node, *Object* node or *Executable* node. Control nodes are *initial* node, *final* node, *decision* node, *merge*, *fork* or *join*. Executable nodes are the directly executable action nodes. The *Call Behavior Action* (CBA) node comes under executable node. The *Value Specification* associated with activity edge involves guard conditions, variable/object values, etc.

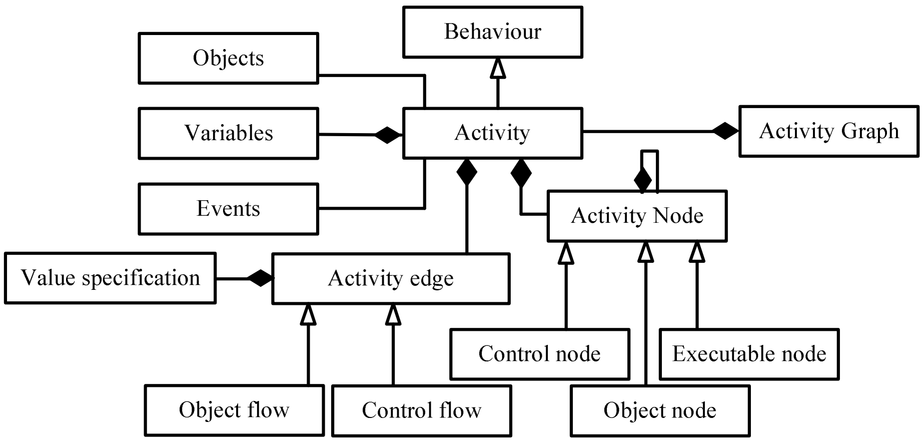


Figure 6.2: Simplified Meta model of UML2.0 Activity Diagram

Table 6.1: Mapping the formalization elements to Metamodel

Element in Metamodel	**Components in Formalization**
Activity Graph	AG_{sd}
ActivityNode	N
ActivityEdge	E
Events	e
ValueSpecification	G
Variable	Var
Objects	o
ControlNode	CN
ExecutableNode	AN_{d}, CBA

6.3 Code Generation Process

The prototype generation process includes four major steps. In the first step, the workflow of a system is modelled using activity and sequence diagrams. The OCL statements are included for adding finer details. In the second step, the diagrams and OCL statements are checked for errors. In the third step, the diagrams and OCL statements are converted to XML files. In the final step, these XML files are converted to prototype of the system. This four step process of code generation is shown in Figure 6.3.

The modeler gives the option to design a system using UML 2.0 activity diagram and sequence diagram. The model processor does the parsing and modification of the object tree. The XML generator regenerates the XML document which contains the modified data. Transformation Engine includes Activity Diagram Transformer and Sequence Diagram Transformer. The Activity Diagram Transformer transforms the activity diagram to the prototype. It is subdivided into three parts; AD parser, AD prototype generator and the transformation rules. The AD parser takes the input as the XML files. It will be parsed to get the details of the activity diagram. These data will be given to the prototype generator.

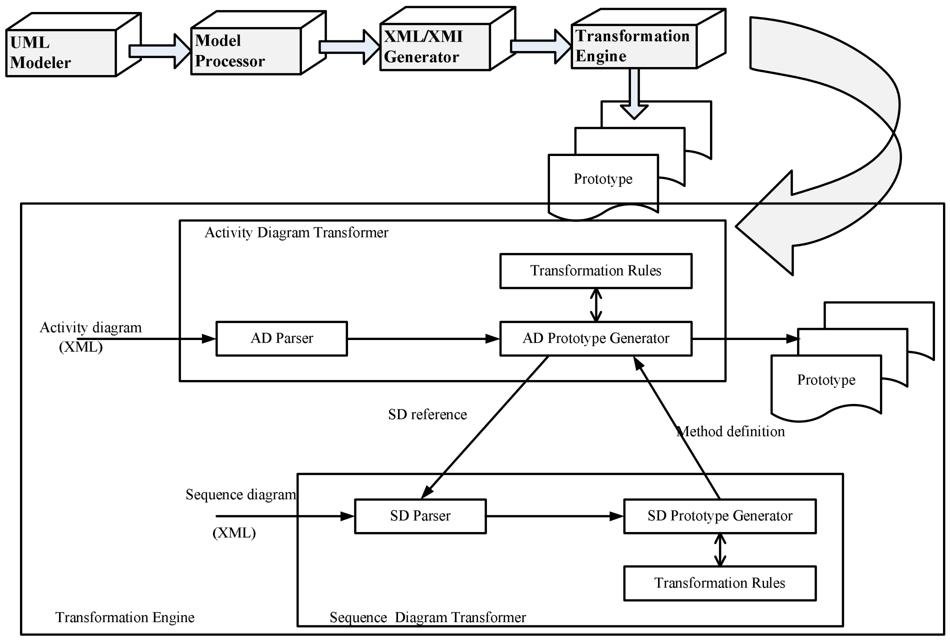


Figure 6.3: Code Generation process

The AD prototype generator applies transformation rules on the activity diagram data to get the prototype. Since the sequence diagrams have been attached to the activity diagram, the prototype generation will not be independent. It has to refer to the sequence diagram details and should be merged with the prototype. The AD prototype generator sends the sequence diagram reference, which has been used in the workflow modeling, to the Sequence Diagram Transformer.

The corresponding sequence diagram tree will be traced out and parsed by the SD parser. The SD parser takes the XML document of the sequence diagram as input. The SD prototype generator generates the method definitions based on the sequence diagram and its transformation rules. The prototype generators (AD prototype generator and the SD prototype generator) use the algorithms which are given in the next section for code generation.

6.4 Code Generation Algorithm

The prototype generation algorithm, *Am_To_Prototype*, takes the object tree of activity graph (AG_{sd}) and sequence diagram (SD) as input and the output is the prototype P of the activity model (AM). We use the term activity model (AM) to refer to the combination of activity and sequence diagram. It generates one Main class and one class for each type of object present in AG_{sd}. Main class includes a main() method which initiates execution. This algorithm uses two sub procedures, *Excecution_Logic* and *Method_Body*. *Excecution_Logic* is used to implement the main() method in the Main class. *Method_Body* is used to implement the definition of all methods in AG_{sd}.

The classes generated after *Am_To_Prototype* is shown in the Figure 6.4. There will be one main class, say the Context class and other associated class corresponds to each object present in the work flow.

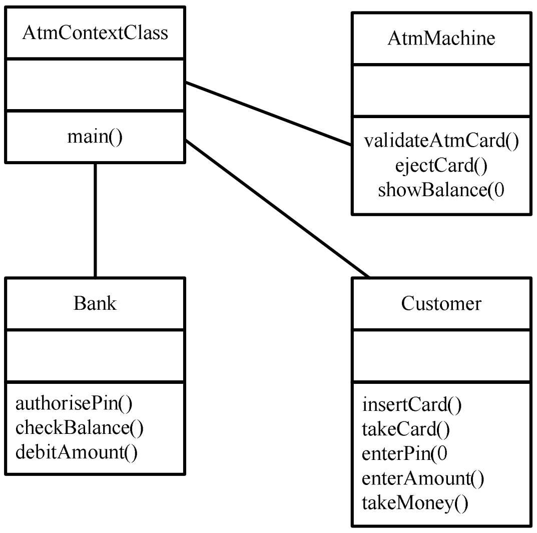


Figure 6.4: Class diagram generated from Am_To_Prototype for Figure 6.7

6.4.1 Algorithm Am_To_Prototype

Am_To_Prototype algorithm first creates a java package with the name of the activity diagram. It applies Depth First Search on AG_{sd} to visit all nodes in the activity graph. A method declaration will be added whenever a new action node is visited. It checks each node in the AG_{sd} and identifies all action nodes and writes the method declaration for all those action nodes. If there is any *callBehaviorAction* node, then call the subroutine *Method_Body*. The algorithm identifies currently active object, ca_obj and its type. The active objects will be fetched from the XML document of AG_{sd}. If no class exists corresponding to the object, then a class will be created. Otherwise the existing class will be updated with the method declarations and added to the prototype P. The search stops when all nodes have been visited. A main class will be created and embed a main() method in it. Then call *Excecution_Logic* subroutine to implement the main() body. The generated class diagram can be edited to add attributes in the classes.

Algorithm 6.1 *Am_To_Prototype*

```
     Input   :    Object tree of $AG_{sd}$ and $SD$
     Output  :    Prototype $P$
1:   create a Java package with the name of $AG_{sd}$
2:   perform Depth First Search on $AG_{sd}$ by keeping initialNode as the
     starting vertex.
         2.1 visit next node $N$
         2.2 identify the currently active object, say ca_obj and its type,
             say $T$
         2.3 check whether class $T$ exists in $P$.
         2.4 if no, add class $T$ to $P$
         2.5 if ($N == AN$)
             add method declaration of nodeName() to class $T$
         2.6 else if ($N == CBA$)
             call subroutine Method_Definition(ca_obj, $N$, $T$, $SD$,
             $P$)
         2.7 Repeat steps 2.1 to 2.6 until all nodes have been visited
3:   Add a Main class to $P$
4:   write first line of main() method of Main class to $P$
5:   call subroutine Excecution_Logic($AG_{sd}$, $P$, main, main, initialNode) to
     define main execution logic of the system
6:   return $P$
```

6.4.2 Algorithm Excecution_Logic(AG_{sd}, P, parentThread, currentThread, start)

The *Excecution_Logic* algorithm generates the execution logic from AG_{sd}. It converts each node in the AG_{sd} to its corresponding programming statements. It considers activity nodes, decision nodes, fork, join, etc., for conversion. The overall AG_{sd} is traversed in depth first search manner starting from the initial node. When it is an action node, corresponding method will be called using the currently active object. The '*if statement*' will be generated for the corresponding decision nodes.

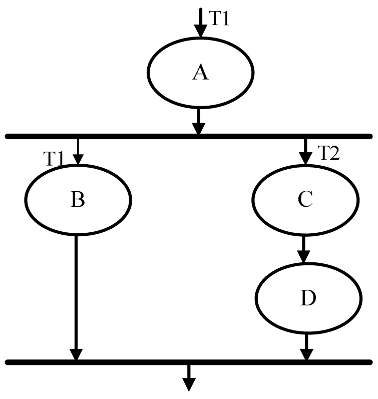


Figure 6.5: Sample fork and join nodes

Fork and join are handled in a different manner. Fork starts some sub threads at that point of execution. Each thread should run concurrently. If fork node has two child nodes one sub thread should be created and started there. Out of two child nodes, one will be the current node itself and the second one is the sub thread.

See Figure 6.5 for sample fork-join segment in an AG_{sd}. Suppose T1 is the current thread and it calls method A(). Then it reaches a fork node. Fork node has two child nodes and one of them is T1 itself and the other one is T2 which is a sub thread of T1. All actions coming under T1 will belong to main() method if T1 is the *main* thread. Otherwise it will belong to the run() method of Thread class. Whenever a new thread is generated it is named after its parent thread suffixed with t_1, t_2, etc. If T1 is the main thread, then T2 will be named as *main_*t_1. These thread names will be pushed to stack for future use. The segment of AG_{sd} between fork and join node is treated separately. *Excecution_Logic*() method will be called recursively for each thread between a pair of fork and join. For Figure 6.5, *Excecution_Logic*() will be called for T1 and T2 after the fork node.

When a join node is encountered by the parent thread (here it is T1), it will just return the recursion. If it is a sub thread, a currentThread.join() statement needs to be added in the parent method. When T2 reaches the join node it adds *main_t_1*.join(); statement to the main() method (assume T1 is the *main* thread. If T1 is not the main thread, then the *main_t_1*.join(); statement needs to be added to the run() method where threadName=name of T1. All these updates will be added to the prototype *P*. when the search reaches the *actionFinal* node, the algorithm returns with the updated prototype *P*. The *Excecution_Logic* algorithm is given below.

Algorithm 6.2 *Excecution_Logic(AG_{sd}, P,* parentThread, currentThread, start*)*

```
Input     :    $AG_{sd}$, P, parentThread, currentThread, start
Output    :    Updated P
Perform Depth First Search on $AG_{sd}$ by keeping start as starting vertex
1.  visit next node N
2.  identify currently active object, ca_obj,
3.  if(N = = AN)
    3.1 create statement ca_obj.actionName();
            write to main() if currentThread='main'
            otherwise write to run() method where
      threadname=currentThread
4.  else if(N = = decision)
    4.1     create "if(guard condition)" statement
            write to main() if currentThread='main'
            otherwise write to run() method where
            threadname=currentThread
    4.2     go to step 1
5.  else if(N = = fork)
    5.1     count the number of child nodes, say c.
    5.2     while ( c>=1 )
            5.2.1  create and start thread with name parentThread _$t_c$,
                write to main() if currentThread='main'
                otherwise write to run() method where
                threadname=currentThread
        5.2.2   push parentThread_$t_c$ to stack
        5.2.3   decrement c by 1
    5.3  if (currentThread = 'main')
        5.3.1   Call Excecution_Logic($AG_{sd}$,$P$, currentThread,
                currentThread, currentForkNode)
        5.3.2   Pop sub threads from stack and call
                Excecution_Logic($AG_{sd}$,$P$,
                currentThread,parentThread_$t_c$, currentForkNode)
                till stack is empty
    5.4     Else if(currentThread ≠ 'main')
        5.4.1   Call Excecution_Logic($AG_{sd}$,$P$, currentThread,
      currentThread, currentForkNode)
        5.4.2   modify run() method in Thread class
        5.4.3   add "if(threadname==parentThread_$t_c$){ "
        statement to run() method
        5.4.4   call Excecution_Logic  ($AG_{sd}$,$P$,currentThread,
                parentThread_$t_c$, currentForkNode)
6.  else if ($N$ = = join)
    6.1     if(currentThread=='main')
            return
    6.2     else if(currentThread==parentThread)
            return
    6.3     else if(parentThread=='main')
            add "currentThread.join()" to main() method
            return
    6.4     else if currentThread ≠ parentThread
            add "currentThread.join()" to run() method where
            threadname=parentThread
            return
7.          else if($N$ = = actionFinal)
```

```
        return P
8.   Repeat steps 1 to 7 until all nodes have been visited
9.   return
```

6.4.3 Algorithm Method_Body (ca_obj, N, T, SD, P)

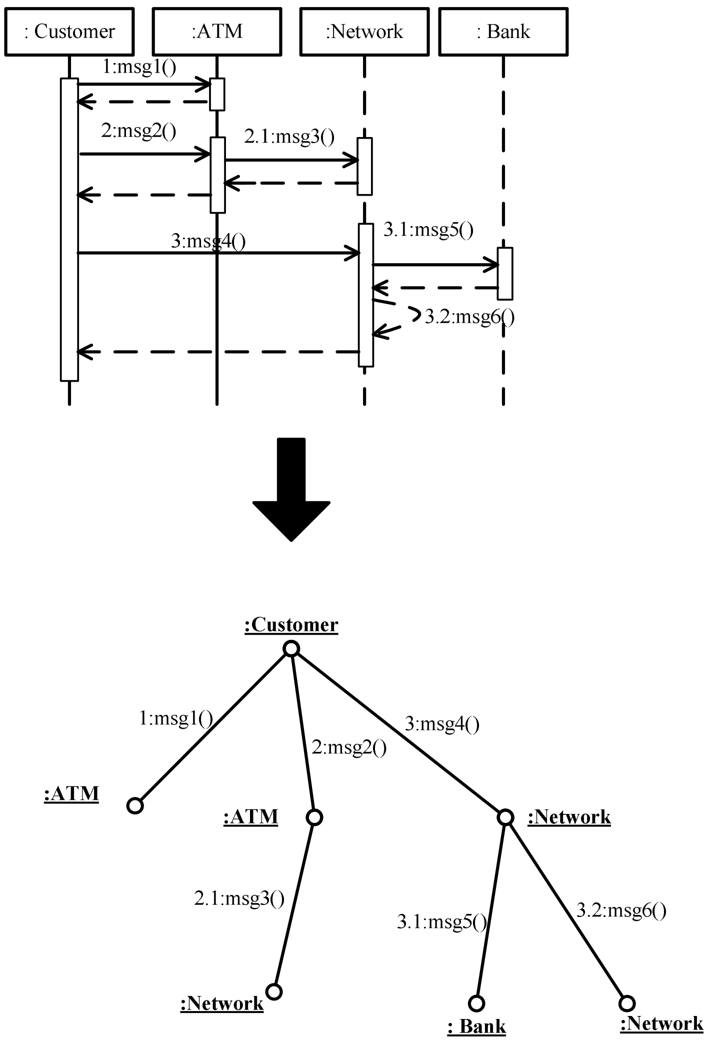


Figure 6.6: A sample sequence tree.

The *Method_Body* algorithm takes current node, currently active object, its class T and sequence diagram tree as input and returns the modified prototype P. Sequence diagram tree is nothing but a tree which is constructed

[104] using object nodes and message flow edges [76]. The SD tree gives a tree representation of the sequence diagrams. This tree representation allows the traversal through the sequence diagram and it is essential when we automate the sequence diagram implementation process.

Each node in the SD tree is the objects that are parts of the collaboration model. Edges represent the messages. The root of the tree will be the object which initiates the collaboration. For instance, there are four objects in a collaboration, say :Customer, :ATM, :Network and :Bank as shown in Figure 6.6. :Customer initiates three message sequences. It is shown as three edges from the customer object. The receiver of the messages is shown at the other end of the edge. For example, the receiver of msg1() is :ATM. It is given as the end node of the first branch from :Customer. The message msg1() gets a return value from :ATM, so that sequence (path in tree) ends at :ATM. Each path in the tree ends when it reaches a node which gives a return for the instantiating message. This tree helps to show the synchronization between messages.

The *Method_Body* algorithm defines the function related to the current activity node N in the activity graph and it will be added to the prototype P. First of all, search the SD tree to find a node with the name of the current object. The number of child nodes will be counted, if the current object is found in the tree. Each edge to the child node is converted to a method invocation statement. It is done for all child nodes and in the order of the message. The Method_Body algorithm is given below.

Algorithm 6.3 *Method_Body (ca_obj, N,* T*, SD, P)*

```
Input  :   current active object, current activity node, T, $SD$, $P$
Output :   method body
1:  Write first line of method definition as "T :: nodeName ( ){ " to $P$
```

```
 2:  Search $\mathcal{SD}$ tree to find a node with nameca_obj.
 3:  If found, count the number of child nodes of ca_obj, say count.
 4:  Take message, msg order, with smallest order   number.
 5:  Write statement obj.action();  to $\mathcal{P}$ , where action is msg.action
     and obj is the receiver node of msg order
 6:  Reduce count by 1
 7:  Remove msg order  from $\mathcal{SD}$  tree
 8:  Repeat steps 4 to 7 until count becomes 0
 9:  Close the method definition
10:  return
```

The working of these algorithms is demonstrated using a case study in the next section. The operation of a bank ATM is taken as the case study.

6.5 ATM Case Study

Consider the work flow of an ATM machine. Figure 6.7 gives a simple activity diagram for ATM transaction. It includes four activities and one decision making. The four activity nodes cause the generation of four method declarations, say, *login()*, *withdraw*(), *updateBalance*() and *eject*() in the class *Atm*. The activities *login* and *withdraw* are expanded using sequence diagrams. These sequence diagrams will fill up the definition of the methods *login*() and *withdraw*().

The activities inside *login* are described in the sequence diagram. It includes PIN verification (*verifyPIN*()), validation and verification of the account with the bank network. The sequence of interactions is shown as method calls. The activity *withdraw* is shown in the second sequence diagram. It includes the selection of the service from the choice menu, verifying the feasibility of the transaction, issue of receipt, etc.

In short, the case study presented here contains one activity diagram ($\mathcal{AG}_{sd}$) and two sequence diagrams ($\mathcal{SD}$) to elaborate the login( ) and withdraw( ) functionalities. The object tree of $\mathcal{AG}_{sd}$ and $\mathcal{SD}$ are the inputs to the algorithm Am_To_Prototype. The following paragraphs show how the algorithms convert the $\mathcal{AG}_{sd}$ and $\mathcal{SD}$ to the prototype.

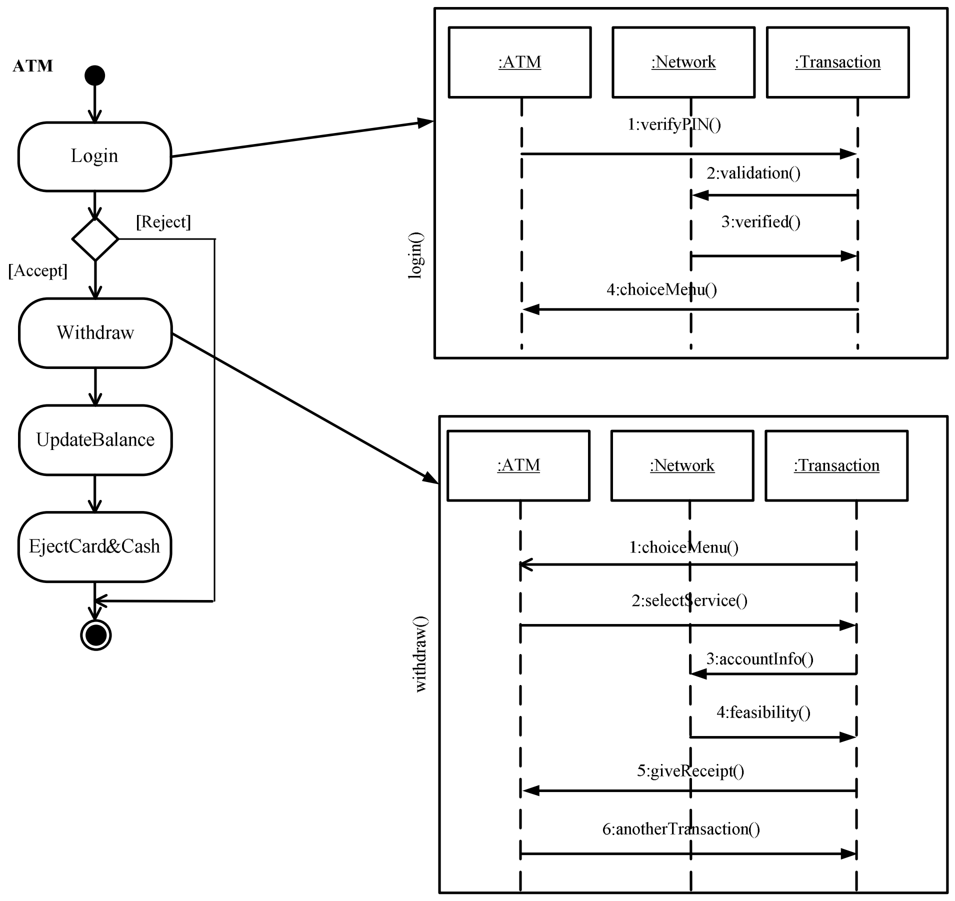


Figure 6.7: ATM Transaction

Code generated by algorithm Am_To_Prototype

The algorithm, *Am_To_Prototype,* will create a package called ATM since $\mathcal{AG}_{sd}$ is named as ATM. Perform DFS on $\mathcal{AG}_{sd}$. Visit Login node. Create a class, Atm, for the current object and add to the package ATM. Call subroutine Method_Body( ) to generate the method definition of Login( ). Visit next nodes and call subroutine Method_Body( ) to generate the method definition of withdraw( ). Add a Main class to the package ATM. Call subroutine Excecution_Logic( ) to generate the definition of main( ) function.

```
/* ========= Generated  Code ================== */
        /* creates package*/
        package AM.myPrototype.ATM;
        /* class definitions*/
        public class Atm
        {       boolean login(PIN);
                void withdraw();
                void updateBalance();
                void ejectCashNCard();
        }
        /* Define Main class */
        Public Main
        {
                /* variable declarations*/
        }
/* =================== Code Ends ================== */
```

Code generated by algorithm Method_Body

The subroutine, *Method_Body()*, will be called twice to generate the definitions for login( ) and withdraw( ). For login( ):- Creates definition of login(). Search *SD* tree to find the node with name *atm*. Identify all messages originating from *atm*. There is only one message *verifyPIN*( ) and prepare a function call statement with the receiver of the message (here it is *transaction*). Similar processing is done for the function withdraw( ).

```
/* ========= Generated  Code ================== */
        /*Method definition for Atm:: login( ) */
        boolean login(PIN ) {
                transaction.verifyPIN(PIN );
        }
        /*Method definition for Atm:: withdraw( ) */
        Atm:: withdraw( ) {
                transaction.selectService( );
                transaction.anotherTransaction( );
     }
/* =================== Code Ends ================== */
```

Code generated by algorithm Excecution_Logic

The subroutine, *Excecution_Logic()*, generates the definition of main() function. The $\mathcal{AG}_{sd}$ is traversed in Breadth First Search manner and find out each node and interprets accordingly. The only object active here is *atm*. So, all the action nodes are called with the object *atm*. For decision making, the guard conditions are checked and true/false paths are traversed. There can be nodes of type action node, initial and final nodes, decision node, fork/join nodes, call behaviour action nodes, etc.

```
/* ========= Generated  Code ================= */
        /* main () method definition */
        Main :: main(String arg[]){
                dn_1=atm.login(PIN );
                if(dn_1){
                        atm.withdraw();
                        atm.ejectCashNCard();
                }
                else
                { exit(0); }
        }
        /*Method definition for Atm:: withdraw( ) */
        Atm:: withdraw( ) {
                transaction.selectService( );
                transaction.anotherTransaction( );
        }
/* ================== Code Ends ================= */
```

To make the description simpler, we omitted the codes that are generated to declare the variables and objects. The additional codes for constructors and destructors also remain hidden in this description.

6.6 Comparison with Related Works

Focus on work flow automation. Code generation from UML models is an interesting research area. However, few works [53, 143] focus on work flow automation. Most of the research works in code generation focus on class diagrams and state machines [29, 49, 55]. We focus on activity diagrams to model work flow. We propose a method to automate work flow/ process flow. We have presented a formal definition for the activity models (work flow models) and collaboration models which reveals the association between the two. The formal definition helps us to device an algorithm to generate software prototype from the UML models.

Prototype generation process. Our work presents a precise description of the prototype generation process. The prototype generation process is enriched with a detailed algorithm based on the formal definitions of AM and CM. Even though few research works [29, 53, 90, 139] give a minimal description of the code generation process, very few works [107] present an algorithm for prototype generation. To the best of our knowledge, no algorithms are reported for code generation that associates AM and CM. We have presented an efficient algorithm for prototype generation in our work, bearing time complexity O (|E|), where E is the number of transitions in the activity diagram. The algorithm works in the depth first search fashion and hence holds the same time complexity.

Degree of automatic code/prototype generation. When we develop a software using object oriented concepts, it includes class declarations, method definitions and the instructions to show control flow. Many of the research works in the field of code generation support automatic generation of class declarations [29, 139]. Some of the research works produce class declaration as well as a part of the method definitions [53, 70, 107, 150]. Few research works provide methods to implement class declarations, control flow and part of the method definition [143, 90]. Still, the code generation of method (or function) definitions remains incomplete. In our work we propose a method to implement method definitions with the help of activity diagram and sequence diagram.

Table 6.2: Comparison with related works

Sl. No	Related Works	[150]	[29]	[139]	[107]	[53]	[70]	[90]	[143]	AutoKode
	Features									
1	**Work flow Automation**	Nil	Nil	Nil	Nil	**yes**	Nil	NIL	**yes**	**yes**
2	**Code Generation from Activity Models with Concurrent actions**	Nil	Nil	Nil	Nil	Nil	Nil	Nil	nil	**yes**
3	**Formal Definition for associating AM with CM**	Nil	Nil	Nil	Nil	Nil	Nil	Nil	nil	**Yes**
4	**Process of prototype generation**	**Yes**	**yes**	**yes**	Nil	**yes**	Nil	**yes**	nil	**Yes**
5	**Efficient algorithm for code/Prototype generation**	Nil	Nil	Nil	**yes**	Nil	Nil	Nil	nil	**Yes**
6	**Degree of automatic code/prototype generation**	Class definition, If..else statements	skeletal code for class definitions	class definition & associations	class declarations, method definitions	control flow	class & method declarations	control flow, object manipulation, user interaction	class declarations, control flow	class declaration, method definition, control flow
7	**Implementation of the method**	**Nil**	**yes**	**yes**	Nil	Nil	**yes**	**yes**	**yes**	**Yes**
8	**MDA approach**	**Yes**	Nil	Nil	Nil	**yes**	**yes**	**yes**	Nil	**yes**
9	**Support for UML2.0**	**Yes**	Nil	**yes**	**yes**	**yes**	**yes**	**yes**	**yes**	**Yes**
10	**Support for Activity diagram**	**Yes**	Nil	Nil	Nil	**yes**	Nil	**yes**	**yes**	**yes**
11	**Support for Sequence diagram**	Nil	Nil	Nil	**yes**	Nil	Nil	**yes**	Nil	**yes**
12	**Programming language supported**	**Java**	**Java**	**Java**	**rCOS**	**BPELWS**	**Java,C++**	**Java**	**java**	**Java**

Table 6.2 gives a summary of the comparison with the related works. Considering the degree of automatic prototype generation, we could find that our method has a clear advantage over existing research works. Our method generates class declarations along with the method definitions, preserving the control flow of the system. Concurrency in activity models is not considered in other publications for code generation.

6.7 Conclusion

The UML activity diagrams help us to generate the source code for the control flow in the system, since it describes how use cases are achieved by providing conditions, constraints and sequential & concurrent activities. The object interactions cannot be generated from the activity diagrams. So, in this chapter we proposed a method to associate UML activity diagram with sequence diagrams to improve the information contained in the activity diagram and thereby improve the generated source code.

We presented a new method to generate Java code from workflow models, which is represented as a set of UML activity diagrams and sequence diagrams. We defined activity diagram and sequence diagram formally which proves the association among both the diagrams. This formal definition helped us to formulate a precise algorithm for code generation. To the best of our knowledge, no other reported research outcome provides both the formal definition and the precise algorithm for code generation combining object interactions and activity.

The proposed algorithms deal with concurrency in the activity diagram which is a big step in research in this area. The algorithms have been implemented, AutoKode, which automatically generates Java source code from the workflow model of a system. Our method ensures more than 80% of completion of the prototype of the system. Since we associate activity models with collaboration models we are able to handle object interactions and are able to generate more complete source code automatically.

Comparison with the related works shows that our approach provides a new method to associate the activity diagram with sequence diagram which improves the quantity and quality of information included in the activity model. Our method provides a new method for work flow automation with the help of a formal definition for activity diagram and sequence diagram association and an efficient algorithm which can deal with concurrent activities in the system models. The degree of automatic prototype generation is better than other related works since our method produces complete class definition, method definition and control flow considering object interactions. Our method will be suitable for current and future software development scenarios since we use MDA approach and supports UML 2.x and machine independent language Java.

Chapter 7 CODE GENERATION FROM STATE CHART MODELS

7.1 Introduction

A software system can be modeled using class diagrams, state chart diagrams and activity diagrams. Software system consists of communicating objects. Each object has its own state transition diagram. State diagram listen for events and act accordingly. The interesting part is that these diagrams can contribute much to the automatic code generation. The system modeled using these three diagrams can be translated to implementation code using code generation tools. Class diagrams help us to generate structural codes and the other two diagrams help us to generate behavioral codes. Out of these three diagrams, UML state chart diagrams are most popular standard for embedded system design [135] and event driven system modeling [75, 137, 44, 136, 39]. This chapter concentrates on state machines since it can generate 100% code out of simple state chart diagrams.

In this chapter we present a method to convert hierarchical states, concurrent and history states to Java code. In our method, we follow a design pattern based approach. A design pattern gives the overall implementation outline using a class diagram [12, 140]. The surveys on code generation from state machines [27] show that the research outcomes are not giving an effective method to implement the concurrent states. We present a design pattern to implement the state hierarchy, concurrency and history state.

The main contributions of the chapter are as follows:

- It presents an easily understandable and reusable design pattern for state machine implementation.
- The design pattern is expandable and is able to handle hierarchical states.
- It gives an effective method to implement composite states with parallel regions in object oriented way.
- It presents a simple method to keep shallow and deep history in a state machine

A state machine can be defined as a graph of states and transitions [62, 43]. State chart diagram may be attached to classes, use cases and collaborations to describe the dynamics of an individual object. It models all possible life histories of an object of a class. Any external influence to the object is called as an event. The response to the event may include the execution of an action and transition to a new state. Events may have parameters that characterize each individual event instance. Inheritance and concurrency can be modeled in state machines. A sample state chart diagram is shown in the Figure 7.1.

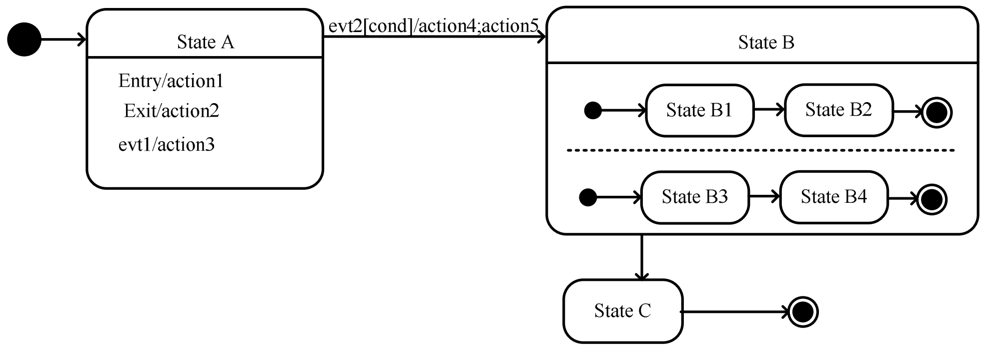


Figure 7.1: State chart essentials

The commonly used features of a state chart diagram are listed as state, transition, event, guard, entry action(s), exit action(s), transition action(s) and internal action inside state

The state of an object is defined as a time period in its life. During this period, the object may wait for some event, or it may perform some activity. There can be named states as well as unnamed (anonymous) states. The transition is the response of the object to an event occurrence. Transition will have an event trigger and a target state. It may include a guard condition and an action. There can be different types of transitions like, entry action, exit action, external transition and internal transition. A Guard condition is a Boolean expression which is evaluated when a trigger event occurs. If the expression evaluates to true, then the transitions occurs. An action is an atomic computation which can be a simple assignment or arithmetic evaluation statement. It can also be a sequence of simple actions. The Entry and Exit actions exist in composite states which contains nested states. Entering the target state executes an entry action and when the transition leaves the original state; its exit action is executed before the action on the transition and the entry action on the new state.

The composite state may contain sequential states (OR type states), or (and) concurrent states (AND type states). The concurrent states form orthogonal regions in the composite states. The states in two orthogonal regions are concurrent states. For example, in Figure 7.1, state B is a composite state which contains two orthogonal regions. In one region, there are two sequential states B1 and B2 and in the other region there are two states B3 and B4. The existence and execution of state B3 and B4 is independent of that of states B1 and B2. In other words, states B3 and B4 are concurrent with states B1 and B2.

In this chapter, we discuss how these elements can be represented in an object oriented program. We assume that the source code has to be generated automatically from the state chart diagram with the help of some CASE tool. The following section examines different methods to map the state chart diagram to program constructs.

7.2 Implementing Hierarchical, Concurrent And History States

In design pattern based approach, each state of the system (or object) is implemented as a class. If the object has three states, then there will be 3 classes, representing each state, in the implementation. Generalization is applied when there are hierarchical states (composite states). For example in Figure 7.2, state A is a composite state. It contains two sub states B and C. So there will be three classes A, B and C. In the implementation pattern they appear as in Figure 7.3. The sub states B and C share the properties of the super state A. So, inheritance (generalization) is the best choice to implement the state hierarchy.

There can be composite states with orthogonal regions. For example, in Figure 7.4 we can see, state A contains 2 sub states B and D. So the implementation contains three state classes, class A, class B and class D. The super state and sub states are implemented using generalization as in Figure

7.5. This generalization can only show that A is a super state and B and D are the sub states. Another important property of the sub states is not addressed here. It is the concurrency between the sub states. State B belongs to one region and state D belongs to another region. That means the existence of state B is independent of that of state D. The state transitions of the state in the orthogonal regions are independent of each other, or in other words, they are concurrent.

To implement the concurrent states, we defined a base class called OrthogonalProperty. According to our new approach, every composite state class has to inherit the properties of the OrthogonalProperty class as in Figure 7.6. The OrthogonalProperty class has two class has two methods to set the number of regions and to execute the sub transistions.

Each composite state has two important attributes to store the number of regions as well as the active sub states. The number of orthogonal regions in the composite state will be stored in the attribute no_of_regions. The active sub states will be registered in the sub state array named as sub_states [ ]. If there are two orthogonal regions, then there can be a maximum of two active sub states. As the number of regions increases, the entries in the sub state array increases. Now, the sub state transitions are implemented based on the sub state array. Whenever there is a transition between sub states, the sub state array will be updated with the new target sub state.

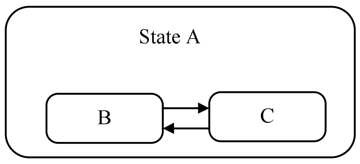


Figure 7.2: Composite state A with two sub states

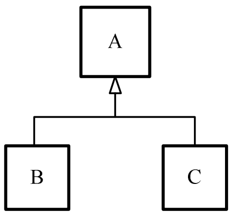


Figure 7.3: Implementation Pattern of the composite state A

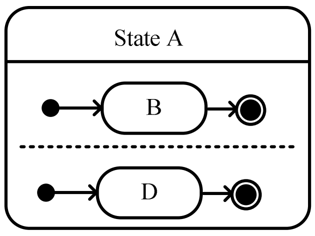


Figure 7.4: Composite state A with two orthogonal regions

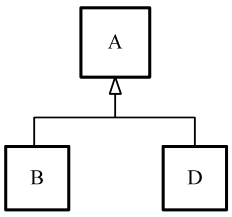


Figure 7.5: Implementation of the composite state A

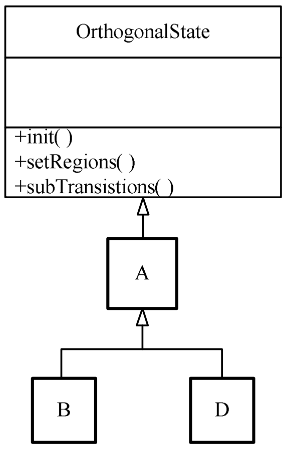


Figure 7.6: Implementation Pattern of the composite state A with orthogonal regions

Next task is to implement history states. There can be two types of history; shallow and deep. Deep history gives the inner (nested) states that were active previously and the shallow history gives the outer state which was active previously. According to the above pattern for orthogonal state, the active state (outer state) is managed by context class and the nested states are managed by the composite class. So it is easy to maintain the shallow history in Context Class and the deep history in CompositeState class.

7.3 Design Pattern for Hierarchical, Concurrent and History States

In this section, we present the design pattern for implementing the UML state chart diagram. In our previous work [121] we have presented a design pattern for concurrent and hierarchical states. Another important feature of state diagram, the state history, is incorporated in the new pattern, named as "Template HHCStateMachine", as shown in Figure 7.7. As per Gamma [37] giving name to a design pattern will help us to refer to the pattern frequently. The pattern gives a simple and easy to use method for state chart implementation using object oriented concepts.

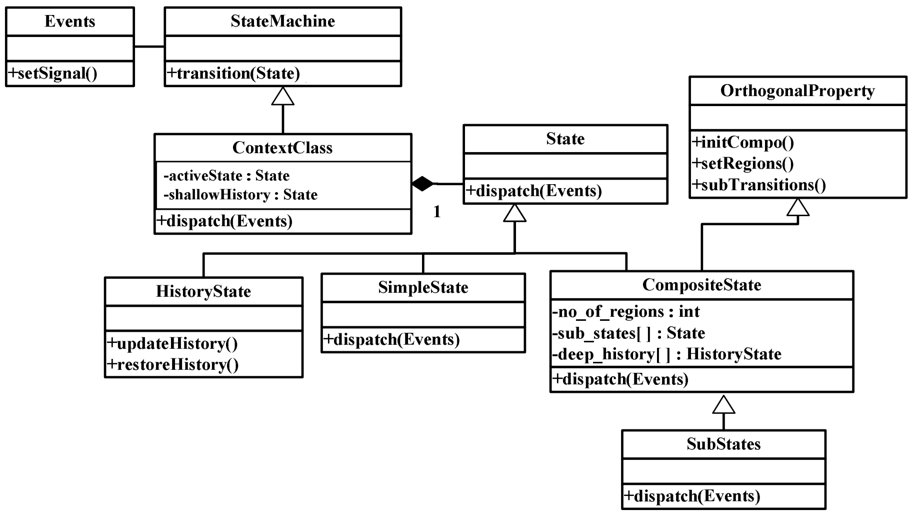


Figure 7.7: The proposed design pattern "Template HHCStateMachine" for state machines

The states and events in a system are implemented as classes in the pattern. There will be events in the system that may or may not change the state of the system. State changing events should initiate state transition. All these terms are included in the StateMachine class which is a blue print of every state machine.

The ContextClass is the class which represents the actual system to be implemented. The currently active state and the shallow history are maintained as the attributes of this class. It has an event dispatch function. This method is used to delegate the events to the corresponding state classes.

State hierarchy is represented using inheritance of state classes. The pattern defines an abstract class called State which acts as the base class for deriving the states in the system. Each state in the system is defined as a derived class of State class. If there is a composite state, all the sub states will be implemented as the derived class of the composite state class. It keeps the semantics of composite state in UML state chart diagram. According to UML, the sub states have the properties of the composite (outer) state. This property can be satisfied by the use of inheritance to derive the sub state classes.

Another important feature of the state chart diagram is parallel regions inside the composite states. The pattern defines a special class called OrthogonalProperty which captures the features of parallel regions in the composite state. It sets the number of regions using the methods setRegions(). The transition between sub states is implemented using the method subTransitions().

Table 7.1: Mapping State machine elements to program constructs

State Machine Element	Program Construct
State	State Class
Transition	Method in StateMachine class
Event	Events class
Entry / Exit Actions	Method in State class
Internal Action	Method in State class
Hierarchical States	Hierarchy of State classes
Concurrent Transitions	Method in the OrthogonalProperty class
Shallow History	Attribute in ContextClass
Deep History	Attribute in CompositeState class

The CompositeState class maintains the properties of the composite states with or without parallel regions. It has three main attributes; no_of_regions, sub_states[ ] and deepHistory. In a composite state there can be one or more regions. It is stored in the first attribute. If there are multiple regions in a composite state, there will be parallel states. These states which are active simultaneously are stored in the second attribute. Before state transition, the old state is stored in the third attribute. The deepHistory[ ] stores the inner states which were active previously. Hence, concurrent states and history are maintained using the CompositeState class.

The mapping of the state machine elements and the Object Oriented Programming constructs is shown in table 7.1.

The skeletal code structure of the pattern is as follows. It includes the classes for Events, State, StateMachine, ContextClass, OrthogonalProperty and CompositeState.

```
public class Events {
   public void setSignal(){
   }
}
public class State {
   public void dispatch(ContextClass cc, Events e){
   }
}
public class StateMachine{
      public void transition(State target){
   }

}
public class ContextClass extends StateMachine {
    State activeState;
    State shallowHistory;
   public ContextClass () {
   }
   public void init(){
   }
   public void dispatch(Events e) {
   }
}
public interface OrthogonalProperty {
     public void init();
     public void subTransition(int region, State target);
     public void setRegions(int no_of_regions );
}
```

```
public class HistoryState extends State{
    public void restoreHistory(){...}
    public void updateHistory(){.....}
}
public class CompositeState extends State implements OrthogonalProperty{
        int no_of_regions;
     State[] sub_states; HistoryState[] deep_history;
  public void init(){
  }
  public void subTransition(int region, State target) {…………  }
  public void dispatch(ConetxtClass context, Events e) {…………  }
  public void setRegions(int no_of_regions) {
        ………………………….   }
}
```

Next section presents a case study to demonstrate the implementation of the design pattern “Template HHCStateMachine”. Different states of an alarm clock are taken as the case study here.

Table 7.2: State transition table of the alarm clock

Current State	Sub State	Events	[guard]	next state
clockON	timekeeping	TICK		timekeeping
		SWITCH_OFF		clockOFF
	alarmON	TICK	curr_time = alarm_time	alarmON
		SWITCH_OFF		clockOFF
		ALARM_SET		alarmON
		ALARM_OFF		alarmOFF
	alarmOFF	ALARM_SET		alarmOFF
		ALARM_ON		alarmON
		SWITCH_OFF		clockOFF
clockOFF		SWITCH_ON		clockON

7.4 Implementing Alarm Clock

We consider the case of an alarm clock. The clock can be in ON state or OFF state. When the clock is ON it can be in two modes simultaneously, timekeeping mode and alarm mode. There are six events in the alarm clock;

TICK, ALARM_ON, ALARM_OFF, SWITCH_ON, SWITCH_OFF and SET. TICK is the advancement of time in seconds. ALARM_ON is to switch on the alarm and ALARM_OFF is to switch off the alarm. SET signal is used to set the alarm time. SWITCH_ON and SWITCH_OFF signals are used to switch on and switch off the clock respectively. When clock is off, the SWITCH_ON event changes the clock state to clockON state.

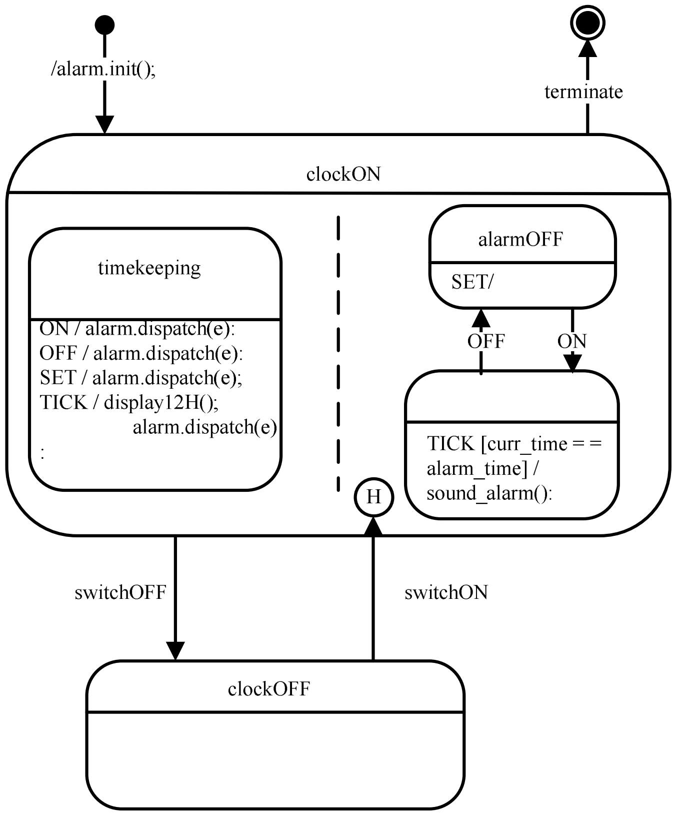


Figure 7.8: UML state diagram representing the Alarm Clock

In the ON state, by default, the clock will be in timekeeping and alarmOFF state. The state changes are shown in Table 7.2. The state diagram of the alarm clock is shown in Figure 7.8. The implementation pattern of the Alarm clock is shown in Figure 7.9.

The implementation of the AlarmClock has 6 state classes. The context class here is the AlarmClock. It uses the Events class and the State class for setting the state of the system and event dispatching. The initial state is set to TimeKeepingState and AlarmOff. Whenever an event encounters the corresponding event handling function will be called by using the run time polymorphism. Whenever the system enters the composite states, the init() function of the class has to be invoked. So this function call is included in the transition function.

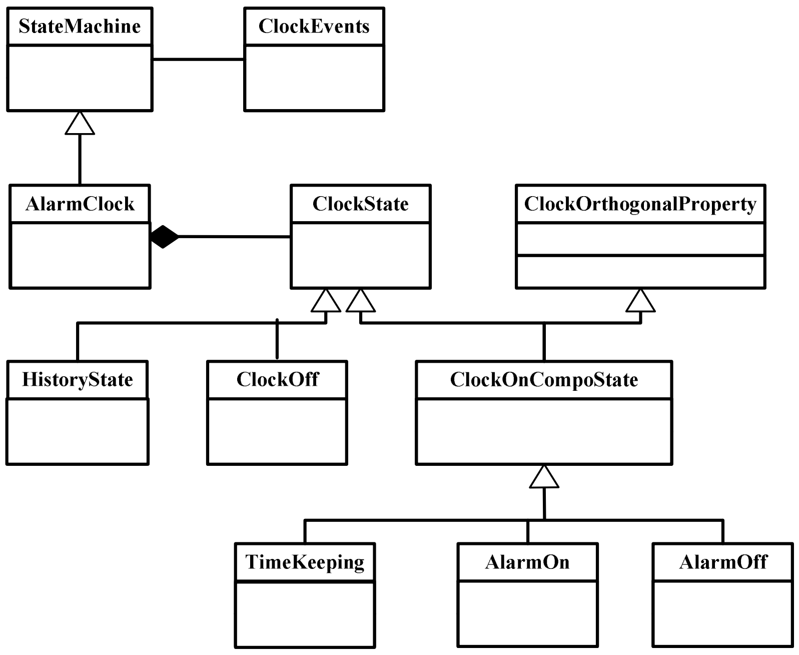


Figure 7.9: Implementation Pattern for the alarm clock

```
public class AlarmClock extends StateMachine {
        ………………………………………………………….
        ………………………………………………………….
        ………………………………………………………….
public static ClockOnCompoState clockOn=new ClockOnCompoState();
    public static ClockOff clockOff=new ClockOff();
    public AlarmClock(int hr, int min, int sec)
    {  curr_time_hr=hr;        curr_time_min=min;        curr_time_sec=sec;
}
    final public void init(){     m_state=clockOff;          tran(m_state);     }
    final public void dispatch(ClockEvents e)
    {            m_state.dispatch(this, e);     }
    final public void tran(ClockState target){
       shallowHistory=m_state;
            m_state=target;
       if(m_state==clockOn)
       {   flag++;
          if(flag==1)            {  clockOn.initCompo();}
          else            { clockOn.restoreHistory();}
        }}
        ………………………………………………………. }
```

The clockOn state is a composite state. It contains two parallel regions; one for time keeping and the other for alarm. It is implemented as ClockOnCompoState. It also implements the sub transition function and history restoring function along with the event dispatch function.

```
public class ClockOnCompoState extends ClockState implements
ClockOrthogonalProperty{
     ……………………………………………………….
     ……………………………………………………….

  public void initCompo()
  {
     deepHistory[1]=timing;
     deepHistory[2]=alarmoff;
     subTransition(1,timing);
     subTransition(2,alarmoff);
  }
  public void subTransition(int region, ClockState target)
  {
     deepHistory[region]=sub_states[region];
     sub_states[region]=target;
  }
   public void dispatch(AlarmClock context, ClockEvents e){
     int i=0;
     for(i=1;i<=no_of_regions;i++){
        sub_states[i].dispatch(context, e);
     }   }
}
```

The clockOff state receives only SWITCH_ON event which causes stat transition to clockOn state. It is implemented in the class ClockOff.

```
public class ClockOff extends ClockState{
   public void dispatch(AlarmClock context, ClockEvents e){
       switch(e.signal){
          case SWITCH_ON :   {context.tran(AlarmClock.clockOn);break;}
       } } }
```

During alarmOff state, the clock receives ALARM_SET, ALARM_ON and SWITCH_OFF events. It is implemented as AlarmOffState class. The alarmOn state is implemented as AlarmOnState class. It accepts TICK, ALARM_SET, ALARM_OFF and SWITCH_OFF events.

ALARM_ON causes sub transition to alarmOn state. During TICK event, the current time is matched with alarm time and if matches it generates alarm sound. ALARM_SET event prompt the user to enter the alarm time. ALARM_OFF event causes sub transition from alarmOn state to alarmOff state.

```
public class AlarmOffState extends ClockOnCompoState{
      Scanner sc= new Scanner(System.in);
   public void dispatch(AlarmClock context, ClockEvents e){
       switch(e.signal){
       case ALARM_SET     :  { System.out.println("Enter Hr : ");
                             context.alarm_time_hr=sc.nextInt();
                              System.out.println("Enter Min : ");
                              context.alarm_time_min=sc.nextInt();
                             System.out.println("Enter Sec : ");
                             context.alarm_time_sec=sc.nextInt();
                             System.out.println("Alarm Set to --->
                             "+context.alarm_time_hr+":"+
                             context.alarm_time_min+":"+
                             context.alarm_time_sec);          break;}
   case ALARM_ON      :  {super.subTransition(2, alarmon);break; }
   case SWITCH_OFF  :  { context.tran(AlarmClock.clockOff);break;   }
}    }  }
public class AlarmOnState extends ClockOnCompoState{
    Scanner sc= new Scanner(System.in);
   public void dispatch(AlarmClock context, ClockEvents e){
        switch(e.signal){
  case TICK      : {
  if((context.curr_time_hr==context.alarm_time_hr)
  &&(context.curr_time_min==context.alarm_time_min)&&
  (context.curr_time_sec==context.alarm_time_sec))
                    { java.awt.Toolkit.getDefaultToolkit().beep(); }
                    break;}
   case ALARM_SET     :  { System.out.println("Enter Hr : ");
                            context.alarm_time_hr=sc.nextInt();
                           System.out.println("Enter Min : ");
                           context.alarm_time_min=sc.nextInt();
                           System.out.println("Enter Sec : ");
                           context.alarm_time_sec=sc.nextInt();
                           break;}
   case ALARM_OFF   :  {   super.subTransition(2, alarmoff);break; }
   case SWITCH_OFF  :  {  context.tran(AlarmClock.clockOff);break;}
  }   }  }
```

The timekeeping state is implemented as TimeKeepingState class. This state is a sub state of the clockOn state. So, the TimeKeepingState is inherited from ClockOnCompoState. This class handles two events ; TICK event and SWITCH_OFF event. TICK event advances the clock time by one second. SWITCH_OFF event causes the state transition to clockOff state.

```
  public class TimeKeepingState extends ClockOnCompoState{
     @Override
     public void dispatch(AlarmClock context, ClockEvents e){
         switch(ClockEvents.signal){
              case TICK      : { AlarmClock.curr_time_sec++;
                      if(AlarmClock.curr_time_sec==60)
                      {  AlarmClock.curr_time_min++;
                         AlarmClock.curr_time_sec=0;
                         if(AlarmClock.curr_time_min==60)
                         {   AlarmClock.curr_time_hr++;
                             AlarmClock.curr_time_min=0;
                             if(AlarmClock.curr_time_hr==13)
                             {   AlarmClock.curr_time_hr=0;
                             }    }  }      break;}
```

```
        case SWITCH_OFF :  { context.tran(AlarmClock.clockOff);
                             break; } } } }
```

The history state is implemented as HistoryState class includes a restoreHistory() function which restores the previous state of the Alarm Clock.

```
public class HistoryState extends ClockState{
   public void restoreHistory(ClockOnCompoState compo)
   {  compo.sub_states[1]= compo.deepHistory[1];
      compo.sub_states[2]= compo.deepHistory[2];
   }}
```

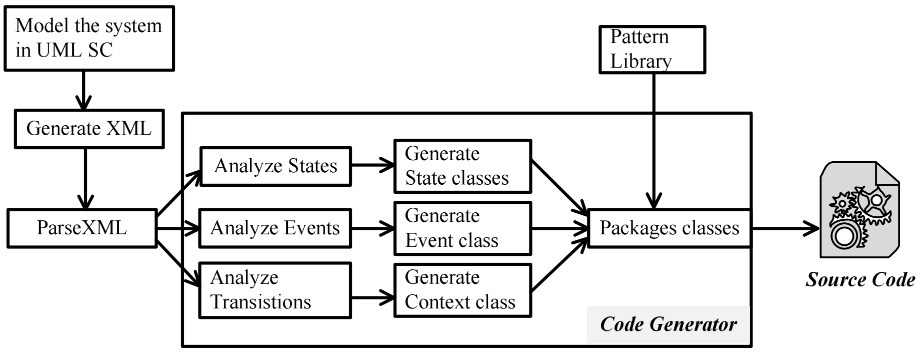


Figure 7.10: Architecture of code generation from state models

7.5 The Code Generation Process

The proposed pattern is saved in the pattern library. This pattern library is used during code generation. The code generation process includes the system modeling in UML state chart diagram, generation of XML for the model, parsing XML and then generating code. The process of code generation is depicted in Figure 7.10.

The representation of the state chart diagram is based on State Chart extensible Markup Language (SCXML) [6]. Based on SCXML, the tags <state>, <transition>, <onentry>, <onexit>, <initial>, <final>, <parallel>, and <history> are used to represent the states, transitions, entry condition, exit condition, initial state, final state, parallel states, and history states respectively. For example, a state with two transitions is given below.

```
<state id=s">
  <transition event="e" cond="x==1" target="s1"/>
          <transition event="e" target="s2"/>
</state>
```

In state ‘s’, if an event ‘e’ occurs and the condition ‘x==1’ is satisfied the state will take transition to state ‘s1’. If the condition is not satisfied, the state will take a transition to the state ‘s2’. The parallel states are represented as follows.

```
<parallel id="p">
<transition event="done.state.p" target= "someOtherState"/>
   <state id="S1" initial="S11">
         ……………………..
         ……………………..
   </state>
   <state id="S2" initial="S21">
         ……………………..
         ……………………..
   </state>
</parallel>
```

Two parallel regions start with sub states S11 and S21. The internal transitions are given between <state> and </state> tags.

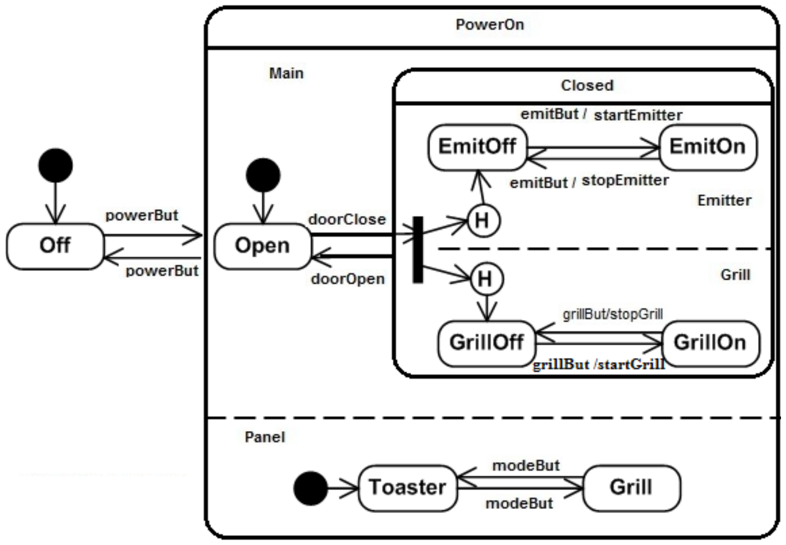
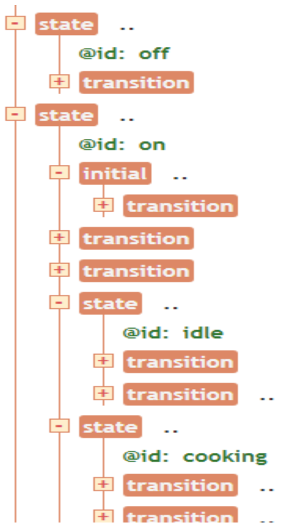


Figure 7.11: UML state diagram representing the microwave oven

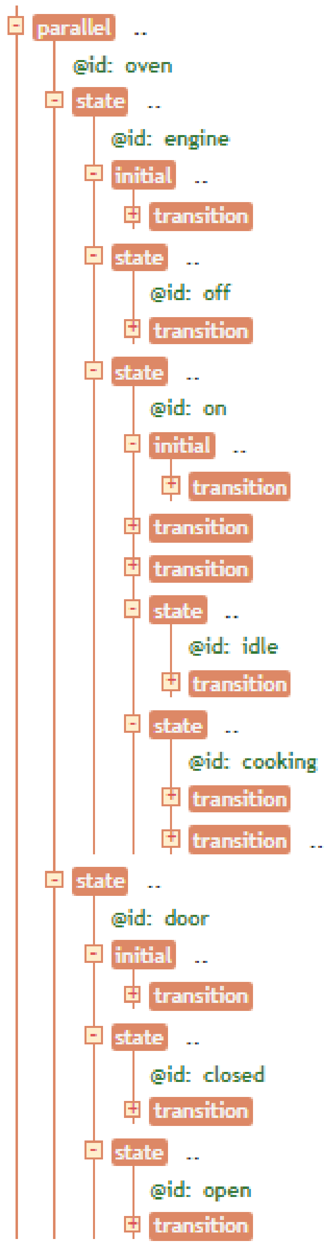
During code generation the parsed XML document is given to the code generator. The code generator has mainly three modules. One module analyzes the parsed XML and find out the state nodes. It generates one state class for each state node based on the design pattern in the pattern library. The composite states are implemented as the extension of State abstract class and the OrthogonalProperty class. The nested states are implemented as the derived classes of the corresponding composite states.



```
- state ..
    @id: off
  + transition
- state ..
    @id: on
  - initial ..
    + transition
  + transition
  + transition
  - state ..
      @id: idle
    + transition
    + transition ..
  - state ..
      @id: cooking
    + transition ..
    + transition ..
```

Figure 7.12: XML representation of Microwave Oven

Consider the state chart diagram of a microwave oven as in Figure 7.11. It includes parallel regions, history states and composite states. The tree view of the XML representation of this is given in the Figure 7.12. It contains two states; off and on. So, two state classes will be generated; offState and onState. The on state contains some sub states. It shows that onState is a composite state. Since <parallel> tag is not there, it is understood that the sub states are not concurrent states. The on state will be defined as OnCompoState class by extending the State class and the OrthogonalProperty class. The off state will be defined as offState class. The inner states idle and cooking will be defined as derived classes of the OnCompoState class; idleState class and cookingState class.



```
⊟ parallel ..
    @id: oven
  ⊟ state ..
      @id: engine
    ⊟ initial ..
      ⊞ transition
    ⊟ state ..
        @id: off
      ⊞ transition
    ⊟ state ..
        @id: on
      ⊟ initial ..
        ⊞ transition
      ⊞ transition
      ⊞ transition
      ⊟ state ..
          @id: idle
        ⊞ transition
      ⊟ state ..
          @id: cooking
        ⊞ transition
        ⊞ transition ..
  ⊟ state ..
      @id: door
    ⊟ initial ..
      ⊞ transition
    ⊟ state ..
        @id: closed
      ⊞ transition
    ⊟ state ..
        @id: open
      ⊞ transition
```

Figure 7.13: XML representation of Microwave Oven with concurrent states

Similarly, the parallel states can be identified by the tag <parallel> as in Figure 7.13. Parallel states are composite states by default. The number of states inside the <parallel> tag will help us to set the number of regions. In the example we have two states inside the <parallel>, that means two parallel regions.

The second module in the code generator analyzes the events and transitions. It generates the event class and updates the event dispatch functions in each state class.

The event is stored as the attribute of the element <transition>. For example, <transition event="e" target="s2"/>. So all <transition> elements have to be checked and list out all those events in the Event class by eliminating the duplications.

```
<state id="S2" initial="S21">
    <state id=S21">
        <transition event="e1" target="S22"/>
    </state>
    <state id="S22">
        <transition event="e2" target="S2Final/>
    </state>
    <final id="S2Final"/>
  </state>
```

In the above XML statements, a composite state with two sub states is given. Two transitions have been described in this composite state. In each <transition> we can identify an event; say e1 and e2. These events will be added to the Events class. If already existing event (in the Event class) is found, it will be ignored.

The event dispatch function will be updated with the transition and the target state. For example, in state S21, there is a transition to state S22 when an event e1 occurs. So, the dispatch function of the class S21State will be updated by adding the corresponding 'case' statement and the state transition with a call to transition() function. The activity inside each state will be converted as the program statements and added to the dispatch function of the corresponding state class.

The third module in the code generator analyzes the state transitions and its flow. It generates the context class for the system. The context class represents the entire system, or the system state chart. It receives the events and delegates the actions to the corresponding active state. The action to be done on a particular transition is defined in the dispatch function of each state class.

The next section presents an evaluation of the code generation method presented in this section. We considered two tools, OCode and Rhapsody for the comparison with our method.

Table 7.3: Efficiency of SMConverter compared with Rhapsody & OCode

	Ocode (ms)	***Rhapsody (ms)***	***SMConverter (ms)***	***Effieciency over Ocode***	***Efficiency over Rhapsody***
Total time for events without transitions	8.8	4.3	3.55		
Average time per event without transition	0.004949	0.002418	0.001997	59.66	17.44
Total time for events having transitions	28.3	19.35	11.35		
Average time per event having transition	0.012736	0.008708	0.005108	59.89	41.34
Total time for all events	37.1	23.65	14.9		
Average time per event	0.009275	0.005913	0.003725	59.84	37

7.6 Evaluation and Comparison with Related Works

The proposed pattern is implemented using the code generator called SMConverter. The performance of SMConverter is compared with other tools like Rhapsody [50] and OCode [56, 55]. We considered the events with and without transitions. Total execution time taken for each type is calculated in milliseconds. Total number of requests for events without transition is 1778 and for events with transition is 2222. The efficiency of our tool (SMConverter) over other tools is shown in the Table 7.3. Figure 7.14 compare the total time taken for events without and with transition respectively.

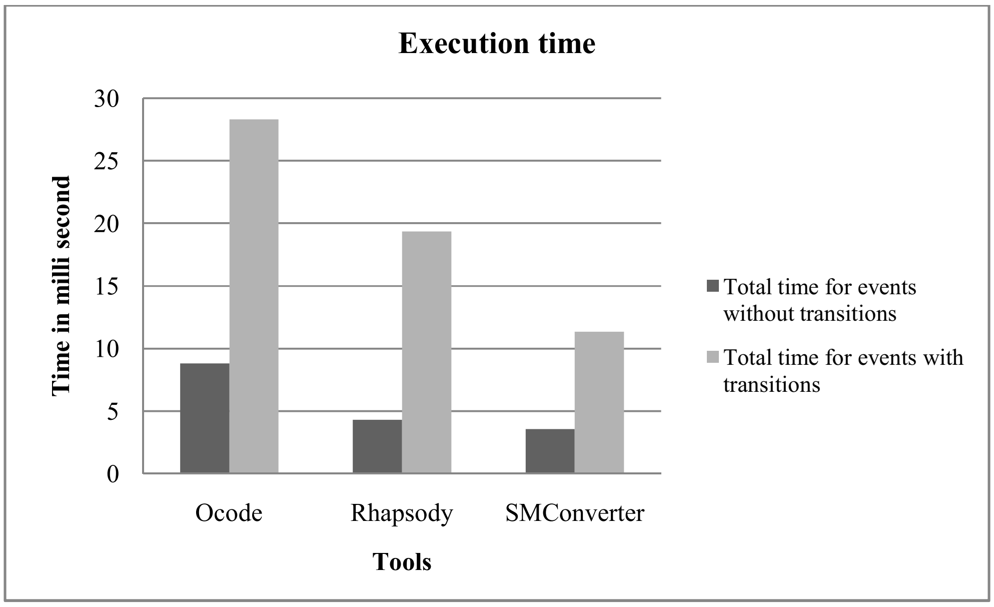


Figure 7.14: Execution time for SMConverter and other tools

The proposed approach is compared with 10 major research works in this area. [94, 2, 56, 49, 55, 131, 78, 89, 111, 132, 133]. For the comparison, we considered the method of implementation of different elements in the state chart diagrams like simple state, composite state, history state etc., and the support for different features of state machines like, hierarchy, concurrency etc. The details of comparison are given in tables 7.4 and 7.5.

7.6.1 Element based comparison

For element based comparison, we considered the basic elements like, current state, state transition etc., in state machines and the components like, orthogonal states, composite states, history states etc., that improve the expressive richness of the state machine. We studied how each of these elements is implemented in the present literatures and how well they support object orientation. The current state, simple states and the context class are represented in a similar way in all the literatures. In case of state transition all literatures except [131] uses switch statement. The components like, composite, orthogonal and history states are supported by few research works.

The element based comparison, given in table 7.4, shows that our method implements the state chart elements in object oriented way, but many of the related works do not.

7.6.2 Feature based comparison

In feature based comparison, we considered the features like, expandability, reusability, understandability etc., and the support for state hierarchy and concurrency in the state chart diagrams.

The state hierarchy is supported by almost all works, except [5], but concurrency is not well supported by the research works. History of state machine state is also not supported by many literatures. Many research outcomes provide methods with good modularity and understandability, but they failed to provide reusability due to the lack of general reusable design pattern. Our proposed method gives a design pattern and which gives good understandability, reusability and expandability.

Feature based comparison, given in table 7.5, shows that our method supports hierarchy and concurrency without spoiling the understandability, reusability and expandability.

7.7 Conclusion

There are different methods for code generation from UML state chart diagrams, like, implementation using switch-case statements, state tables and design patterns. Design pattern approach is widely accepted because of it supports object oriented design and implementation. Moreover, the design

Table 7.4: Element based comparison with related works

Elements\ Reference	**[94,2,56,49,55]**	**[132,133]**	**[131]**	**[89]**	**[78]**	**[111]**	**Proposed method**
Context Class	class	class	subclass	sub class	sub class	class	sub class
Current State	attribute	attribute	attribute	attribute	attribute	attribute	attribute
Simple State	separate class for each state	defining a single state class for all states	separate class for each state	defining a single state class for all states	member functions for each state	defining a single state class for all states	separate class for each state
state transition process	switch stmt	switch stmt	double-dispatch	switch stmt	switch stmt	switch stmt	switch stmt
simple Composite state	derived class of abstract state class	member functions	derived class of abstract state class	a class state for all states	member functions	a template class	derived class of Orthogonal State class (no of regions = 1)
Orthogonal Composite state	derived class of abstract state class	thread	A class for all orthogonal states, derived from abstract state class	X	X	X	derived class of Orthogonal State class
Shallow History	attribute	attribute	attribute	attribute	X	X	attribute
Deep History	X	attribute	attribute	X	attribute	X	attribute

Table 7.5: Feature based comparison with related works

Features\ Reference	[94, 2, 56, 49, 55]	[132, 133]	[131]	[89]	[78]	[111]	Proposed Method
Hierarchy	supported	supported	*supported	supported	supported	supported	supported
Concurrency	supported	supported	supported	not supported	not supported	not supported	supported
History	partially supported	supported	supported	partially supported	partially supported	not supported	supported
Expandability	good	bad	bad(no pattern)	good	medium	medium	good
Simplicity	yes		no	neutral	yes	neutral	yes
Reusability	medium		bad(no pattern)				good
Understandability	good	medium	medium	medium	good	medium	good
Modularity	good	medium	good	good	good	medium	good

pattern approach is easily expandable due to its modular structure. The use of design pattern in code generation improves the quality of the generated code.

The design patterns available in the literatures mainly support hierarchical states in the UML state chart diagrams. The more essential features like concurrent states and history states are not addressed in those design patterns.

In this chapter, we introduced a design pattern based implementation of state machine with hierarchical, concurrent and history states. The design pattern proposed in the chapter provides modularity, reusability and understandability. Moreover it keeps the semantics of state hierarchy and concurrency and history state as well. The code generator presented in the chapter gives a systematic way of code generation from the UML state chart diagrams with the help of the proposed design pattern.

Comparison with related works shows that the proposed method of state chart diagram implementation is a better way to support the important features like concurrency and history states. Moreover, the qualitative comparison with the related works shows that our method supports state concurrency and history without compromising the expandability, reusability and understandability, whereas other methods compromise the above qualities.

Comparison with other tools shows that our method is more efficient in terms of the time taken for event processing. The case study and comparison with other tools reveals that the proposed approach gives less complex code and promising results.

Chapter 8 CONCLUSION AND CONTRIBUTIONS

8.1 Overview

This thesis focuses on an essential requirement in the Software industry, the automation of software development process. The automation will considerably reduce the effort and time required for the software development. Automatic code generation from the system designs is a major mile stone to automation in software development. The behaviour of a system can be designed using UML behavioural models. Many industries like IBM, Microsoft etc and many researchers are interested in generating code from UML models [35]. It is evident from the literature that the current technologies do not provide precise methods for code generation. So the methods for automatic code generation from UML behavioural models are addressed in this thesis.

8.2 Contributions

This thesis proposed methods to generate source code automatically from UML behavioural models. There is no one to one mapping between the behavioural models and the object oriented programming constructs. So far, there is no standard method in the literature to convert the UML behavioural models to source code. In this study, we proposed novel approaches to automatically generate source code from the important behavioural models like use-case diagram, sequence diagram, activity diagram and state chart diagram. These methods will help the software industry to reduce their effort and time, spend on software development. The contributions of the thesis are listed as follows.

1. ***Modularization of Code with UML Use Case Models***

   Specifying the requirements flawlessly is very essential to save time and money in software development. In UML Use case diagram specifies the functional requirements. Moreover, the use cases allows the modularization of the source code based on the services (use cases) provided by the software system. The sequence diagram gives the object interactions to achieve the use case. This thesis proposed a method to generate Java code from the UML use case diagram elaborating the scenarios with sequence diagrams. This prototype code generation helps the software development team to fine tune the customer requirements thereby reduce the flaws in the later stages of the software development.

2. ***Enhancing UML activity models with OCL.***

   The activity diagram gives the sequence of activities, condition and the objects involved in the control flow to achieve a use case. The finer details like, the constraints on activities, initial values for the attributes, parameters to the activities etc have to be added to the implementation code by the developer during the software implementation phase. This thesis proposed a novel approach to enhance UML activity models with OCL so that the code generated from the activity diagram will contain these finer details and thereby reduces the rework during the implementation phase. When we enhance an existing model, it should be stated and proved theoretically. The possibilities to enhance UML activity diagrams with OCL are stated using meta models in this thesis. The thesis introduced several meta models for enhancing activity diagram with OCL. This helps in specifying pre/post conditions on operations and methods, the actual parameters that are passed to the operations, the initial values of the attributes and the guard conditions. This gives a strong foundation to improve the information (or details) given in an activity models and thereby support the code generation from the activity models.

3. ***Operational Semantics for OCL enhanced activity model***

   The formal semantics of a model gives the characteristics and behaviour of the model. No formal semantics for OCL enhanced activity diagram was available in the literature. This thesis presented the formal semantics for OCL enhanced UML activity diagrams. The structural semantics describes the components in the OCL enhanced UML activity diagrams and the operational semantics describes the behaviour of the same.

4. ***Enhancing UML activity models with object interaction details.***

   The activity in the UML activity diagram may include interactions between different objects. These interaction details cannot be modeled using the activity diagram. We have introduced a new method for associating the sequence diagram with complex activities which helps to solve this deficiency. In UML, each diagram is defined independent of each other. So, we have formally defined the association of activity diagram with sequence diagram. This formal definition specifies how the activity and sequence models can be combined and thereby improving the code generation.

5. ***Design pattern for UML state chart diagram with hierarchical, concurrent and history states.***

   UML state chart diagrams shows the state changes of an event driven system. The design patterns specify how to implement a state chart diagram. The available design patterns do not support the concurrent states and the history states. This thesis proposed a new design pattern for state machines with hierarchical, concurrent and history states. This design pattern gives an approach for code generation. Since it is reusable and expandable, more features can be easily added to the pattern during the future research work.

6. ***Automatic code generation process from UML behavioural models.***

   Code generation is a step by step process. In this thesis, we have presented methods for Java code generation utilizing the UML use case models, sequence models, activity models and state chart models.

The static structure of the system can be designed using UML class diagrams. The UML class diagrams can be directly converted to source code in any object oriented language. The programming constructs in object oriented languages allows one to one mapping with each and every element of the class diagram.

The behaviour of a system can be designed using UML behavioural models like use case diagram, sequence diagram, activity diagram and state chart diagram. Unlike the class diagrams, the behavioural models do not have one to one mapping with the programming constructs. There is no straight forward approach to convert these models to source code. So, different researchers follow different methods for code generation from the UML behavioural models. During the literature survey, we have come across some shortcomings in the present methods of code generation. One most important issue is that we cannot generate complete source code from any one of the behavioural models. It is required to add additional information in these diagrams to assist code generation. Chapters 5 and 6 introduced the methods to enhance the UML activity diagram with OCL and sequence diagram. The operational semantics for OCL enhanced activity diagrams is also presented in chapter 5.

The existing code generation methods, as described in the literature review, have its own advantages and limitations. The diagrams used for system design depends on the kind of software we develop. The percentage of code generated in each method varies depends on the features that are considered for the code generation. The main drawback in the existing research works is the lack of algorithms for code generation. Moreover, some features of the diagrams, like concurrent activities, parallel and history states etc, are not considered for code generation in these methods. These features are explored in this research work to generate more lines of code than the existing methods. The code generation from use case and sequence diagrams helps the Business Analyst to generate system prototype and thereby fine tune the user requirements. The chapters 4, 5 and 6 of this thesis presented algorithms for code generation from use case diagram, sequence diagram and activity diagram.

In the case of UML state chart diagrams, few researchers have been working on the code generation from it. These literatures do not present a design pattern for UML state chart diagram with hierarchical, concurrent and history states. The design pattern allows us to generate the source code from the state chart diagrams. This thesis presented a design pattern for the UML state chart diagrams which contains hierarchical, concurrent and history states, in chapter 7.

The aim of the study was to establish code generation approaches for UML behavioural models. The thesis also proposed methods to associate different system models (use case, activity, sequence and state machine) and associating constraint specification languages like OCL with these models for improved code generation. We examined each models separately and investigated for possible combinations of these models to get better results. Operation semantics for such combinations were designed and well explained in the thesis.

These methods provide a novel approach for automatic code generation from the UML behavioural models. It gives guidelines to the CASE tool developers on how to use UML behavioural models for code generation. It will, in turn, be very much useful for the software industry to handle the complexity of the software systems. These approaches will have their reflections in all phases of the software development, till the maintenance phase.

8.3 Future Scope

In the present study, the code generation from the UML Activity Diagrams got improved with the help of OCL. It helps to add more implementation specific details in the activity models. We would like to extend this work to other behavioural models like sequence diagram and state chart diagram.ie, improving the code generation from the sequence and state chart diagrams by enhancing them with the OCL statements.

REFERNENCES

[1] A.Jakimi, M Elkoutbi, “Automatic Code Generation From UML State chart”, International Journal of Engineering and Technology, vol. 1, no.2, June 2009, pp. 165-168

[2] Ali, Jauhar, and Jiro Tanaka, "Converting statecharts into Java code", 5th International Conference on Integrated Design and Process Technology (IDPT’99), Texas, USA, May, 17, 1999.

[3] Alvarez, María Luz, et al, "A Methodological Approach to Model-Driven Design and Development of Automation Systems." IEEE Transactions on Automation Science and Engineering, vol. 15, no.1, June 16 2016, pp. 67-79.

[4] Asma Charfi, et.al, “Does Code Generation Promot or Prevent Optimizations?”, 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 5-6 May 2010. pp. 75 – 79.

[5] B. P. Douglass, “Real Time UML – Developing Efficient Objects for Embedded Systems”, Massachusetts: Addison-Wesley, 1998.

[6] Barnett, J., et al. "State chart XML (SCXML) state machine notation for control abstraction. W3C Recommendation." (2015).

[7] Beckert, Bernhard, Uwe Keller, and Peter H. Schmitt, "Translating the Object Constraint Language into first-order predicate logic", in the Proceedings of the Second Verification Workshop: VERIFY'02, Copenhagen, Denmark, July 25--26, 2002, S. Autexier and H. Mantel, Eds. DIKU technical reports, vol. 02, no. 07, July 2002, pp. 113--123.

[8] Beierlein, Thomas, Dominik Fröhlich, and Bernd Steinbach, "Model-driven compilation of UML-models for reconfigurable architectures",

2nd RTAS Workshop on Model-Driven Embedded Systems (MoDES'04), Toronto, Canada, May 25-28, 2004.

[9] Bendraou, Reda, et al, "A comparison of six uml-based languages for software process modeling", IEEE Transactions on Software Engineering, vol 36 no.5 pp. 662-675, September 2010.

[10] Benjamin Davison, Tom Ruckle, et. al, "Automated Code Generators", Department of Computer Science, University of Minnesota, 2006.

[11] Bichler, Lutz, "A flexible code generator for MOF-based modeling languages", 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture. October, 2003.

[12] Bruegge, Bernd, and Allen H. Dutoit, "Object-Oriented Software Engineering Using UML, Patterns and Java", Prentice Hall, Munich, Germany, 2004.

[13] Cabot, Jordi, and Martin Gogolla, "Object constraint language (OCL): a definitive guide", Formal methods for model-driven engineering. Springer Berlin Heidelberg, June 18-23, 2012, pp. 58-90.

[14] Caplinskas, Albertas, and Johann Eder, eds, "Advances in Databases and Information Systems", 5th East European Conference, ADBIS 2001, Vilnius, Lithuania September 25-28, 2001 Proceedings. Vol. 2151. Springer, 2003.

[15] Charfi, Asma, Chokri Mraidha, and Pierre Boulet, "An Optimized Compilation of UML State Machines", IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. IEEE, 11-13 April 2012.

[16] Chiorean D., Petrascu V., Petrascu D, "How My Favorite Tool Supporting OCL Must Look Like", in the Proceedings of the 8th Inter. Work. on OCL Concepts and Tools (OCL'08) at MoDELS, 2008.

[17] Chiorean, D., Bortes¸, M., Corut¸iu, D, “Proposals for a Widespread Use of OCL”, in: the proceedings of the MoDELS’05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica October 4, 2005.

[18] Chitra, M. T., and Elizabeth Sherly, "Refactoring sequence diagrams for code generation in UML models", Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on. IEEE, September 24-27, 2014.

[19] Correa, Alexandre, and Cláudia Werner, "Refactoring object constraint language specifications", Software & Systems Modeling, vol. 6,. no.2, June 2007, pp. 113-138.

[20] Cristian Georgescu, “Code Generation Templates Using XML and XSL”, C/C++ Users Journal - Mixed-language programming, ACM, vol. 20 no. 1, January 2002, pp. 6-19.

[21] Cyprian F. Ngolah and Yingxu Wang, “Exploring Java Code Generation Based on Formal Specifications in RTPA”, Canadian Conference on Electrical and Computer Engineering vol. IV, 2004, pp. 1533-36.

[22] D. Kundu, D. Samanta, and R. Mall, “Automatic code generation from unified modelling language sequence diagrams”, IET Software , vol.7 , no. 1, pp. 12-28, February 2013.

[23] Dang, Duc-Hanh, and Martin Gogolla, "An OCL-Based Framework for Model Transformations", VNU Journal of Science: Computer Science and Communication Engineering, vol. 32, no.1, March 2016.

[24] Dang, Duc-Hanh, Anh-Hoang Truong, and Martin Gogolla, "Checking the Conformance between Models Based on Scenario Synchronization", J. UCS vol. 16, no. 17 2010: pp. 2293-2312.

[25] Daw, Zamira, and Rance Cleaveland, "An extensible formal semantics for UML activity diagrams", arXiv preprint arXiv:1604.02386, April 2016.

[26] E. Dominguez et.al, "A Systematic review of code generation proposals from state machine specifications", Journal of Information and Software Technology, vol. 54, no. 10, October 2012, pp. 1045-1066..

[27] E. Dominguez et.al. "A Systematic review of code generation proposals from state machine specifications", Journal of Information and Software Technology, vol. 54, no.10, October 2012 pp 1045-1066.

[28] Eikermann, Robert & Hölldobler, Katrin & Roth, Alexander & Rumpe, Bernhard, "Reuse and Customization for Code Generators: Synergy by Transformations and Templates", 6th International Conference, MODELSWARD 2018, Funchal, Madeira, Portugal, January 22-24, 2018.,pp. 34-55

[29] Eivind Bjoraa, Torgeir Myhre, Espen Westlye Straapa, "Generating Java Skeleton From XMI", Open Distributed Systems, Paper presented on Open Distributed Systems, Agder University College, 2000.

[30] EL B. Omar, B. Brahim and G. Taoufiq, "Automatic code generation by model transformation from sequence diagram of systems internal behavior", International Conference on Information Technology and Communication Systems, vol. 1, no. 2, 2012.

[31] Eshuis, Rik, and Roel Wieringa, "A formal semantics for UML Activity Diagrams-Formalising workflow models", Centre for Telematics and Information Technology (CTIT), CTIT Technical Report Series, vol.1, no. 4, February 2001, p. 44.

[32] Flake, S, “Enhancing the Message Concept of the Object Constraint Language”, In SEKE vol. 4, June 2004, pp. 161-166.

[33] France, Robert B., et al, "Model-driven development using UML 2.0: promises and pitfalls", Computer, vol. 39, no. 2, Feb. 2006, pp. 59-66.

[34] G. Booch, J. Rumbaugh, and I. Jacobson. “The Unified Modelling Language User Guide” Addison-Wesley Object Technology Series, January 1999.

[35] G.K.A. DIAS, "Evolvement of Computer Aided Software Engineering (CASE) Tools: A User Experience", International Journal of Computer Science and Software Engineering (IJCSSE), Volume 6, Issue 3, March 2017, ISSN (Online): 2409-4285 pp. 55-60

[36] Gall, Dariusz & Walkowiak, Anita, "An Approach to Semantics for UML Activities", Advances in Intelligent Systems and Computing · September 2018, pp. 252-262.

[37] Gamma, Erich, “Design patterns: elements of reusable object-oriented software”, Pearson Education India, Edition. 1, November 2015.

[38] Gomaa, Hassan, and Erika Mir Olimpiew, "The role of use cases in requirements and analysis modeling", in the proceedings of the 2nd International Workshop on Use Case Modeling (WUsCaM-05): Use Cases in Model-Driven Software Engineering, Montego Bay, Jamaica, October 2-7 2005.

[39] Gonzalez, Ariel & Luna, Carlos & Cuello, Roque & Pérez, Marcela & Daniele, Marcela, "Towards an automatic model transformation mechanism from UML state machines to DEVS models", CLEI electronic journal [online], vol.18, no.2, August 2015, pp.4-4.

[40] Gotti, Sara, and Samir Mbarki, "UML executable: A comparative study of UML compilers and interpreters", 2016 International Conference on Information Technology for Organizations

Development (IT4OD), USMBA University, Fez, Morocco, March 30 - April 1st, 2016. IEEE, 2016.

[41] Hamie, A, “Translating the object constraint language into the java modelling language”, in Proceedings of the 2004 ACM symposium on Applied computing, Nicosia, Cyprus, March 14 - 17, 2004, pp. 1531-1535.

[42] Hamie, A., Howse, J., & Kent, S, “Interpreting the object constraint language”, in the proceedings of the 5th Asia-Pacific Software Engineering Conference (APSEC '98), Taipei, Taiwan, ROC, December 2-4 1998, IEEE Computer Society 1998, pp. 288-295.

[43] Harel, David, and Amnon Naamad, "The STATEMATE semantics of statecharts", ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 5 no.4, October 1996, pp. 293-333.

[44] Harel, David, and Orna Kupferman, "On object systems and behavioral inheritance", in IEEE Transactions on Software Engineering, vol. 28, no. 9, September 2002, pp. 889-903.

[45] Harrand, Nicolas, et al, "ThingML: a language and code generation framework for heterogeneous targets", in the proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-malo, France, October 02 - 07, 2016, pp. 125-135.

[46] Hitz, Martin, and Gerti Kappel, "Developing with UML – Some Pitfalls and Workarounds.” In: Bézivin J., Muller PA. (eds) The Unified Modeling Language. «UML»’98: Beyond the Notation. UML 1998, Springer, Berlin, Heidelberg 1998, Lecture Notes in Computer Science, vol. 1618,

[47] Hussmann, Heinrich, and Steffen Zschaler, "The object constraint language for UML 2.0–overview and assessment", Upgrade

Journal, vol.5, no.2, February, 2004.

[48] I. Jacobson, . Rumbaugh, and G. Booch, “The Unified Modelling Language Reference Manual”, Addison-Wesley, 1999.

[49] Iftikhar Azim Niaz, “Automatic Code Generation From UML Class and State chart Diagrams”, Thesis Report, University of Tsukuba, Japan, 2005.

[50] I-Logix Inc. Rhapsody. Accessed: Mar. 2010. [Online]. Available:http://www.ilogix.com

[51] Imran Sarwar Bajwa, M. Imran Siddique, M. Abbas Choudhary, ”Rule based Production Systems for Automatic Code Generation in Java”, Digital Information Management, 1st International Conference, Bangalore, India, December 06-08, 2006, pp.300 – 305.

[52] J Cabot et. Al, “Verification of UML/OCL Class Diagrams using Constraint Programming”, Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW’08, Lillihammer, Norway, 9-11 April, 2008.

[53] J Kochler, R Hauser, S Sendall, M Wahler, “Declarative techniques for model-driven business process integration”, IBM Systems Journal, Volume 44, No 1, pp. 47-65, 2005.

[54] J. Ali, “Using Java Enums to implement Concurrent-Hierarchical State Machines”, Journal of Software Engineering, vol. 4, no. 3, October, 2010, pp. 215-230

[55] J. Ali, and J. Tanaka, “An Object Oriented Approach to Generate Executable Code from OMT-Based Dynamic Model”, Journal of Integrated Design and Process Science, vol. 2, no. 4, 1998, pp. 65-77.

[56] J. Ali, and J. Tanaka, “Implementing the Dynamic Behavior Represented as Multiple State Diagrams and Activity Diagrams”,

Journal of Computer Science & Information Management (JCSIM), vol. 2, no. 1, 2001, pp. 24-36.

[57] J. Breti, “State Machine Code Generation in Python”, document version 1.0.1, Gnosis Version, Canada, 2007.

[58] J. Rumbaugh, I. Jacobson, and G. Booch, “The Unified Software Development Process”, Addison-Wesley, 1999.

[59] Jacobson, Ivar, “Object-oriented software engineering: a use case driven approach”, Pearson Education India, 1993.

[60] Jakimi, A., El Koutbi, M, “An object-oriented approach to UML scenarios engineering and code generation”, International Journal of Computer Theory Engineering, vol. 1, no.1, 2009, pp. 35–41.

[61] James Rumbaugh, Ivar Jacobson, Grady Booch, “Object-Oriented Analysis and Design with Applications, Third Edition ”, Addison-Wesley, 2007.

[62] James Rumbaugh, Ivar Jacobson, Grady Booch, “The Unified Modeling Language Reference Manual”, Addison-Wesley, 1999.

[63] Jiang, Ke, Lei Zhang, and Shigeru Miyake, "An executable UML with OCL-based action semantics language", 14th Asia-Pacific Software Engineering Conference (APSEC'07), Aichi, December 04 – 07, 2007, pp. 302-309.

[64] Jon Siegel, “Introduction to OMG’s Model Driven Architecture”, Object Management Group, 2004.

[65] Jurack, Stefan, et al. "Sufficient criteria for consistent behavior modeling with refined activity diagrams", Model Driven Engineering Languages and Systems, LNCS vol. 5301, 2008, pp. 341-355.

[66] Kamalrudin, Massila, John Hosking, and John Grundy, "Improving requirements quality using essential use case interaction patterns", in the Proceedings of the 33rd International Conference on Software

Engineering. ACM, Waikiki, Honolulu, HI, USA, May 21 - 28, 2011, pp. 531-540.

[67] Kevin Lano, “The UML-RSDS manual”, Technical report, Department of Informatics, King’s College London, May 2014.

[68] Kleppe, Anneke, Jos Warmer, and Steve Cook, "Informal formality? the Object Constraint Language and its application in the UML metamodel", International Conference on the Unified Modeling Language. Springer Berlin Heidelberg, June 3-4, 1998.

[69] Knieke, Christoph, et al, "Defining domain specific operational semantics for activity diagrams", Technical Report, Institut für Informatik, Technische Universität Clausthal, Zellerfeld, Germany, vol.12, no.04, December 2012.

[70] Kurt T Rudhal, Sally E Goldin, “Adaptive multi-language code generation using YAMDAT”, Proceedings of ECTI-CON 2008, Proceedings of the 5th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, 2008.Volume 1, May 2008, pp. 181 – 184 .

[71] Kyas, Marcel, and Frank S. de Boer, "On message specifications in OCL", Electronic Notes in Theoretical Computer Science vol. 101, November 2004, pp. 73-93, Available: 10.1016/j.entcs.2004.02.017.

[72] Lano, Kevin, "A compositional semantics of UML-RSDS." Software and Systems Modeling, vol. 8 no.1 February 2009, pp. 85-116.

[73] Lano, Kevin, and Shekoufeh Kolahdouz-Rahimi, "Model-transformation design patterns." IEEE Transactions on Software Engineering, vol. 40, no.12, 2014, pp. 1224-1259.

[74] Lano, Kevin, and Shekoufeh Kolahdouz-Rahimi, "Specification and verification of model transformations using UML-RSDS" International Conference on Integrated Formal Methods,

Nancy, France, 11 - 14 October 2010, Lecture Notes in Computer Science, vol. 6396. Springer, Berlin, Heidelberg.

[75] Lethbridge, Timothy Christian, and Robert Laganiere, Object-oriented software engineering. New York: McGraw-Hill, 2005.

[76] Li, Xiaoshan, Zhiming Liu, and He Jifeng, "A formal semantics of UML sequence diagram", 2004 Australian Software Engineering Conference. Proceedings, Melbourne, Victoria, Australia, 2004, pp. 168-177.

[77] Ljubica Lazareviae, Dragan Miliaev, "Finite State Machine Automatic Code Generation", IASTED conference, Austria, 2000.

[78] M. Samek, "Practical Statecharts in C/C++, Quantum Programming for Embedded Systems", CMP Books, 2002.

[79] M.H Aabidi et.al. "An Object Oriented Approach to generate Java code from hierarchical- concurrent and history states", International Journal of Information and Network Security (IJINS), vol-2, no. 10, December 2013, pp 429-440.

[80] Madhusudhan Govindaraju, "XML Schemas Based Flexible Distributed Code generation Framework", IEEE International Conference on Web Services (ICWS 2007), Salt Lake City, UT, 2007, pp. 1212-1213. doi: 10.1109/ICWS.2007.199.

[81] Mage, Kjetil. "A Pratical Application of the Object Constraint Language OCL", 2002.

[82] Maryam Jamal, Nazir Ahmad Zafar, "Formal Semantics of Executable Node and Activity Group of UML 2.5 Activity Diagram", International Conference on Communication Technologies (ComTech), Rawalpindi, 19-21 April 2017, pp. 174-179

[83] Mathupayas Thongmak, Pornsiri Muenchaisri, “Design of Rules for Transforming UML Sequence Diagrams into Java code”, Ninth Asia-Pacific Software Engineering Conference, 2002., Gold Coast, Queensland, Australia, 2002, pp. 485-494.

[84] Maylawati, Dian & Darmalaksana, Wahyudin & Ramdhani, Muhammad, "Systematic Design of Expert System Using Unified Modelling Language", IOP Conference Series: Materials Science and Engineering, vol. 288, January 2018, pp. 1-7

[85] Maylawati, Dian & Ramdhani, Muhammad & Syakur Amin, Abdusy, "Tracing the Linkage of Several Unified Modelling Language Diagrams in Software Modelling Based on Best Practices", International Journal of Engineering and Technology, UAE, vol. 7, 2018, pp. 776-780.

[86] Mehjabin Pathan, Aakash Patkar, Sayali Surve, "Automatic Partial Code Generation Using Class and Sequence Diagrams", International Journal on Recent and Innovation Trends in Computing and Communication, Volume: 4 Issue: 3 ISSN: 2321-8169, March 2016, pp-365 - 368

[87] Mellor, J. Stephen, Balcer, M., and Foreword By-Jacoboson, I, “Executable UML: A foundation for model-driven architectures”, Addison-Wesley Longman Publishing Co., Inc.., 2002.

[88] Milicev, Dragan. "Automatic model transformations using extended UML object diagrams in modeling environments", in IEEE Transactions on Software Engineering, vol. 28, no. 4, pp. 413-431, April 2002..

[89] Miro Samek, “Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems", Newnes, Newton, MA, 2008.

[90] Muhammad Usman, Aamer Nadeem, Tai-hoon Kim, "UJECTOR: A tool for Executable Code Generation from UML Models", 2008 Advanced Software Engineering and Its Applications, Hainan Island, 2008, pp. 165-170.

[91] N. Debnath, A. Funes, A. Dasso, G. Montejano, D. Riesco and R. Uzal, "Integrating OCL expressions into RSL specifications," 2007 IEEE International Conference on Electro/Information Technology, Chicago, IL, 2007, pp. 158-162.

[92] N. S. Bhullar, B. Chhabra and A. Verma, "Exploration of UML diagrams based code generation methods," 2016 International Conference on Inventive Computation Technologies *(ICICT)*, Coimbatore, 2016, pp. 1-6.

[93] Nanthaamornphong, Aziz & Leatongkam, Anawat, "Extended ForUML for Automatic Generation of UML Sequence Diagrams from Object-Oriented Fortran", Scientific Programming, Febraury 2019, pp. 1-22.

[94] Niaz, Iftikhar Azim, and Jiro Tanaka, "An object-oriented approach to generate Java code from UML statecharts." International Journal of Computer & Information Science, vol. 6, no.2, 2005, pp. 315-321.

[95] Nick Fargo, "State.js", MIT, 2013. Last Accessed: January 2019, "http://statejs.org/docs/"

[96] Object Management Group Standard, "Action Language For FoundationalUML - ALF," Available: http://www.omg.org/spec/ALF/1.0/ Beta2 [Accessed: April 15, 2013].

[97] "MDA Guide Version rev 2.0", OMG Document ormsc/2014-06-01, OMG, June 2014. https://www.omg.org/cgi-bin/doc?ormsc/14-06-01. Last accessed on 11-04-2019.

[98] "Object Constraint Language", OMG Available Specification, Version 2.4, February 2014.

[99] OMG Unified Modeling Language (OMG UML), Infrastructure, V2.5, [Online]. Available: http://www.omg.org/spec/UML. Dec 2017

[100] Paetsch, Frauke, Armin Eberlein, and Frank Maurer, "Requirements engineering and agile software development." WET ICE 2003, Proceedings, Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003., Linz, Austria, 2003, pp. 308-313.

[101] Pandey, R. K, "Object constraint language (OCL): past, present and future." ACM SIGSOFT Software Engineering Notes, vol. 36, no. 1, 2011, pp. 1-4.

[102] Parada, A.G., Siegert, E., de Brisolara, L.B.: 'Generating Java code from UML class and sequence diagrams'. Proc. of the 2011 Brazilian Symp. on Computing System Engineering, 2011, pp. 99–101

[103] Philip Samuel and R. Mall, "Slicing-based Test Case Generation from UML Activity Diagrams", ACM SIGSOFT Software Engineering Notes, vol.34, no. 4, 2009, pp. 1-14.

[104] Philip Samuel, Rajib Mall, Pratyush Kanth, "Automatic test case generation from UML communication diagrams", Elsevier, Science Direct, Journal of Information and Software Technology, vol. 49, no. 2, pp. 158–171, February 2007.

[105] Philip Samuel, Sunitha E V, "Automatic Code Generation using Model Driven Architecture", Proceedings of 2009 IEEE International Advance Computing Conference (IACC 2009), Patiala, India, March 2009, pp. 2339 – 2344,.

[106] Philip Samuel, Sunitha E.V, "Document Type Definition for the XMI Representation of UML2.0 Activity Diagram", International Journal

of Recent Trends in Engineering, vol. 1, no. 1, May 2009, pp. 206 – 210.

[107] Q.Long, Z.Liu et.al., “Consistent Code Generation from UML Models”, 2005 Australian Software Engineering Conference, Brisbane, Queensland, Australia, 2005, pp. 23-30. doi: 10.1109/ASWEC.2005.17.

[108] Roger S Pressman, “Software Engineering: A Practitioner's Approach”, 7th edition, McGraw-Hill, 2014.

[109] Ruben Campos, “Model Based Programming: Executable UML with Sequence Diagrams”, CS Thesis, California State University, Los Angeles, 2007

[110] S Sengupta et. al, “Automated Translation of behavioral models using OCL and XML”, T ENCON 2005 - 2005 IEEE Region 10 Conference, Melbourne, Qld., 2005, pp. 1-6.

[111] S. Heinzmann, “Yet another hierarchical state machine”, Association of C & C++ Users, Overload Journal no. 64, December 2004, pp. 14–21.

[112] S. Maoz, J. O. Ringert, and B. Rumpe. “An Operational Semantics for Activity Diagrams using SMV”, Technical Report AIB, 2011-07, RWTH Aachen University, Germany, January 2011.

[113] Sami Beydeda , Volker Gruhn, “Model-Driven Software Development”, Springer-Verlag New York, Inc., Secaucus, NJ, 2005.

[114] Scott W. Amble, “Agile Modeling: A Brief Overview”, Presented at the Workshop of the pUMLGroup held together with the «UML»2001 on Practical UML-Based RigorousDevelopment Methods - Countering or Integrating the eXtremists, Toronto, Canada, 01, October, 2001.

[115] Sendall, Shane, and Wojtek Kozaczynski, “Model transformation the heart and soul of model-driven software development”, in IEEE Software, vol. 20, no. 5, pp. 42-45, Sept.-Oct. 2003.

[116] Smialek, Michal, Norbert Jarzebowski, and Wiktor Nowakowski, "Translation of use case scenarios to Java code", Computer Science, vol. 13, no. 4, 2012, pp. 35-52.

[117] Solomencevs, Arturs, "Comparing Transformation Possibilities of Topological Functioning Model and BPMN in the Context of Model Driven Architecture", Applied Computer Systems, vol.19, no. 1, May 2016, pp. 15-24.

[118] Sunitha E.V, Philip Samuel, “Automatic code generation using unified modeling language activity and sequence models”, in IET Software, vol. 10, no. 6, December 2016, pp. 164-172.

[119] Sunitha E.V, Philip Samuel, “Enhancing UML Activity Diagrams using OCL”, Proceedings of the 2013 IEEE International Conference on Computational Intelligence and Computing Research, Enathi, 2013, pp. 1-6.

[120] Sunitha E.V, Philip Samuel, “Translation of behavioral models to source code”, 2012 12th International Conference on Intelligent Systems Design and Applications (ISDA), CUSAT, Kochi, India, November 2012, pp 598-603.

[121] Sunitha, E. V., and Philip Samuel, "Object Oriented Method to Implement the Hierarchical and Concurrent States in UML State Chart Diagrams" Software Engineering Research, Management and Applications. Springer International Publishing, May 2016, pp. 133-149.

[122] Syriani, Eugene, and Hüseyin Ergin, "Operational semantics of UML activity diagram: An application in project management", 2012

Second IEEE International Workshop on Model-Driven Requirements Engineering (MoDRE), Chicago, IL, 2012, pp. 1-8.

[123] T Stahl, M Voelter, K Czarnecki, “Model-Driven Software Development: Technology, Engineering, Management”, John Wiley & Sons, July 2006 , ISBN:0470025700

[124] Tamas Vajk and Gergely Mezei, “Incremental OCL to C# Code Generation”, 2010 International Joint Conference on Computational Cybernetics and Technical Informatics, Timisoara, 2010, pp. 277-280.

[125] TC Lethbridge, R Laganiere, "Object-Oriented Software Engineering Practical software development using UML and Java", Tata McGraw-Hill Education, 2004.

[126] Tim Schattkowsky, Wolfgang Muller, “Transformation of UML State Machines for Direct Execution”, 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, TX, USA, 2005, pp. 117-124.

[127] Tomas G Moreira, et. Al, “Automatic code generation for embedded systems: from UML specifications to VHDL Code”, 2010 8th IEEE International Conference on Industrial Informatics, Osaka, 2010, pp. 1085-1090.

[128] U. A. Nickel, J. Niere, R. P. Wadsack, A. Zundorf, “Roundtrip Engineering with FUJABA”, in the proceedings of 2nd Workshop on Software-Engineering, Bad Honnef, Germany, 2000.

[129] “Object Management Group Standard, Semantics of a Foundational Subset for Executable UML Models”, Version 1.2.1., 2013. [Online]. Available: http://www.omg.org/spec/FUML/

[130] Usman, M., Nadeem, A. “Automatic generation of Java code from UML diagrams using UJECTOR”, International Journal of Software Engineering and its Applications, vol. 3, no. 2, May 2009, pp. 21–38.

[131] V. Spinke, “An object-oriented implementation of concurrent and hierarchical state machines”, Journal of Information and Software Technology, vol. 55, no. 10, October 2013, pp. 1726-1740.

[132] Van Cam Pham, Ansgar Radermacher, Sebastien Gerard and Shuai Li, "Complete Code Generation from UML State Machine", 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017), France, 2017, pp. 208-219

[133] Van Cam Pham, Ansgar Radermacher, Sébastien Gérard, "From UML State Machines to code and back again!", in the Proceedings of the Conference on Computer Science and Information Systems ACSIS, France, vol. 9, 2016, pp. 283–290.

[134] Vaziri, M., & Jackson, D, “Some Shortcomings of OCL, the Object Constraint Language of UML”, TOOLS '00 Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00), vol. 34, July 2000, pp. 555-562.

[135] Wagner, Flávio R., and Luigi Carro, "Embedded SW Design Space Exploration and Automation using UML-Based Tools", Embedded System Design: Topics, Techniques and Trends, Springer US, 2007. 437-440.

[136] Wang, Jiacun, "Finite-State Machines", in book Real-Time Embedded Systems, Wiley Publications, July 2017, pp. 179-195

[137] Wang, Jiacun, ed, “Handbook of Finite State Based Models and Applications”, First Edition, CRC Press, November 2016.

[138] Warmer, Jos B., and Anneke G. Kleppe, "The Object Constraint Language: Precise Modeling With Uml, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999

[139] William Harrison, Charles Barton, Mukund Raghavachari, “Mapping UML designs to Java”, in the proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Minneapolis, Minnesota, USA , vol. 35, no. 10, Oct 2000, pp: 178 - 187.

[140] Wolfgang, Pree, “Design patterns for object-oriented software development, Reading, Mass”, Addison-Wesley, 1994.

[141] X Li, Z Liu, “Prototyping System Requirements Model”, Journal of Electronic Notes in Theoretical Computer Science (ENTCS), vol. 207, April 10, 2008

[142] Yilong Yang, Xiaoshan Li, Zhiming Liu, Wei Ke, Quan Zu, and Xiaohong Chen, "Automated Prototype Generation from Formal Requirements Model",Computing Research Repository (CoRR) in arXiv, August 2018, pp. 1-23

[143] Yin, Ling, Jing Liu, and Zuohua Ding, "Modeling and Prototyping Business Processes in AutoPA," Theoretical Aspects of Software Engineering (TASE), 2011 Fifth International Conference on Theoretical Aspects of Software Engineering, Xi'an, Shaanxi, 2011, pp. 169-176.

[144] Federico CiccozziMälardalen University, Västerås, Sweden “Unicomp: a semantics-aware model compiler for optimised predictable software”, Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, Gothenburg, Sweden — May 27 - June 03, 2018, pp. 41-44

[145] Andrén, Filip, Thomas Strasser, and Wolfgang Kastner. "Engineering smart grids: Applying model-driven development from use case design to deployment", Energies, vol. 10, no. 3, 2017, pp. 374.

[146] Cunha, Alcino, Ana Garis, and Daniel Riesco. "Translating between Alloy specifications and UML class diagrams annotated with OCL." Software & Systems Modeling, vol. 14, no. 1, 2015, pp. 5-25.

[147] Rumpe B. “Agile Modeling with UML: Code Generation, Testing, Refactoring”, Springer International Publishing, Germany, 26 April 2017.

[148] Fouad T, Mohamed B, “Transforming XML Schema Constraining Facets And XML Queries To Object Constraint Language (OCL)”, Journal of Theoretical & Applied Information Technology, vol. 87, no. 3, 31 May 2016.

[149] Pauker F, Wolny S, Fallah SM, Wimmer M. “UML2OPC-UATransforming UML Class Diagrams to OPC UA Information Models”, Procedia CIRP, vol. 67, 1 Jan 2018, pp. 128-33.

[150] Perera, Nipuni. "Automatic conversion of activity diagrams into flexible smart home apps." PhD diss., Auckland University of Technology, 2018.

LIST OF PUBLICATIONS

Journal Publication:

1. Sunitha E.V, Philip Samuel, "*Automatic code generation using unified modeling language activity and sequence models*." IET Software, IEEE Xplore Digital Library, vol.10, no. 6, 2016, pp. 164-172.
2. Sunitha E.V, Philip Samuel, "*Object constraint language for code generation from activity models*", Journal of Information and Software Technology, Elsevier, vol. 103, 2018, pp. 92–111.
3. Sunitha E.V, Philip Samuel," *Automatic Code Generation From UML State Chart Diagrams*", IEEE Access Journal, IEEE Xplore Digital Library, vol. 7, 2019, pp. 8591 - 8608..
4. Sunitha E.V, Philip Samuel, "*Translation of Behavioral Models to Java Code and Enhance With State Charts* ", International Journal of Computer Information Systems and Industrial Management Applications. Dynamic Publishers, Inc., USA, vol. 6, 2014, pp. 294 - 304.
5. Sunitha E.V, Philip Samuel, "Code generation from Use case and sequence diagrams", (communicated).

Conference Publication:

1. Sunitha E.V, Philip Samuel, "*Object Oriented Method to Implement the Hierarchical and Concurrent States in UML State Chart Diagrams*." Software Engineering Research, Management and Applications. Springer, 2016. pp.133-149.
2. Sunitha E.V, Philip Samuel, "*Enhancing UML Activity Diagrams using OCL*", IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), IEEE Xplore Digital Library, Tamilnadu. 2013.
3. Sunitha E.V, Philip Samuel, "*Translation of Behavioral Models to source code*", 12th International Conference on Intelligent Systems Design and Applications (ISDA) CUSAT, IEEE Xplore Digital Library, 2012.

Other Publications:

1. Text Book on "Object Oriented Programming – Basic Concepts", Sunitha EV, Jyothis Publishers. 2011. ISBN -978-93-5351-261-3