AutoCodeApp

[1]Mr. Japan Chetankumar Gor, [2]Mr. Vinay Nagin Panchal, [3]Ms. Siddhi Jayesh Shah, [4]Mrs. Poonam Milind Thakre

[1,2,3]Student, [4]Faculty, Universal College of Engineering Vasai, India, [1]japangor@gmail.com, [2]vinay.npanchal@gmail.com, [3]siddhishah231415@gmail.com, [4]poonam.thakre@universal.edu.in

Abstract— Much of the changes that take place in the world of software hinge on APIs (Application Programming Interfaces). Although, we interact with APIs every day, many still underestimate their impact on web applications. An API is software programming written to bridge the communication between web applications. Hence, it makes services accessible to outside developers so they can build those services into their programs. There is a need to re-imaging APIs in the current scenario. Traditionally, APIs always followed the request-response paradigm. That is, an application sends requests to an API, and the API sends a response back to the application. What we need in this case is a quick way, where APIs themselves are auto-generated, Auto Code Generation existed only in the Bottom most category in the early period, for creating code from Objects during the Object-Oriented era and later on, when Object-Oriented Code became common, Auto code was for Optimizing or Modifying the program Code. Our project sets the future of Automatic Code generation, by using pre-defined templates, taking inputs from the user, and suggesting recommendations for better performance. We will be using Python to automatically generate API using predefined schema and templates.

Keywords — API, Auto, Code, Generation, No code, Program logic, Template, Schema, UML.

I. INTRODUCTION

Certain services help non-programmers to make apps and websites without implementing a single line of code. What if a scenario arises wherein the user wants both of them? There is no such viable ecosystem to maintain and generate data for multiple apps on different platforms without coding. So, we have taken the initiative to fill this gap between website builders and app builders by providing a back-end builder on which users can attach multiple apps. So, we have developed a simple API builder with a drag and drop UI.

The objective of this paper is to develop an API from pre-defined templates, that will be sufficiently well-defined and unambiguous. Our entire project is divided into three parts: - Webapp: Simple click and drag UI for non-programmer users; Communicator: The one which will send and receive data, to and from the web app. And create instructions for the logic executer; Logic executor: Will manipulate pre-created templates and write code for the user based on instructions[5].

II. LITERATURE SURVEY

Jeffrey Smith, Mieczyslaw Kokar, Kenneth Baclawski, "Formal Verification of UML Diagrams: A First Step Towards Code Generation". UML diagrams can be used for code generation. Such code should carry the meaning embedded in a diagram. The goal of this paper is to show a process in which such translation can be formally verified. To achieve this goal, the whole checking process has to be formalized. In this paper, we show such a verification process and example. UML diagrams are translated into code by various CASE tools. However, the verification of the translation correctness is left to either the tool developers or the programmers, CASE tools don't enforce a complete set of UML syntax, let alone semantics and CASE tools are only capable of translating header files and constructors/destructors to a programming language. We are interested in translators that are provably correct concerning the intended meaning of the UML language, i.e., such that preserve the intended meaning embedded in UML diagrams representing various program specifications.[1]

Gergely Pintér, István Majzik, "Automatic code generation based on formally analyzed UML state chart models" The code generation is based on extended hierarchical automata, the formal description method used as an intermediate representation of statecharts for model checking purposes, this way enabling automatic implementation of formally analyzed models. Since state charts can automatically be mapped to extended hierarchical automata, a code generator based on our pattern could be used as a module that can be inserted into any UML modeling tool equipped with model export

 DOI : 10.35291/2454-9150.2022.0141

capabilities. This approach enables the modeler to use the usual design environment and hides the transformation required for model checking and code generation steps.[2]

Ivo Damyanov, Nick Holmes, "Metadata Driven Code Generation Using .NET Framework" In this paper incorporating manual and automatic code generation is discussed. A solution for automatic metadata-driven code generation is presented illustrated with a multi-tier Enterprise Resource Planning System. We intend to make our solution available to the public to encourage investigation of code generation and schema-driven tools for .NET Framework. .NET Framework provides a rich set of language extensions (like Custom Attributes) and namespaces (like System.CodeDom) that make possible automatic code generation. Our approach was proved by developing a complete ERP system with: an entirely auto-generated client facade – about 36,000+ lines of C# code, partially generated (90%) Server and Ether tier code – about 66,800+ lines of C# code, and about 11,800+ lines of SQL code (stored procedures and triggers).[3]

Ingo Stürmer, Mirko Conrad, "Code Generator Certification: A Test Suite-oriented Approach" Since this code is often deployed in safety-related environments, the quality of the code generators is of high importance. The use of test suites is a promising approach for gaining confidence in the code generators' correct Functioning. This paper gives an overview of such a practice-oriented testing approach for code generation tools. The main application area for the testing approach presented here is the testing of optimizations performed by the code generator. The test models and corresponding test vectors generated represent an important component in a comprehensive test suite for code generators. As regards the actual certification situation for development tools, the test suite proposed could be a substantial contribution to current certification practices.[4]

Michael G. Hinchey, James Rash, and Christopher A. Rouff, "Towards a Fully Formal Approach to Automatic Code Generation". In this, a method to mechanically transform system requirements into a provably equivalent model has yet to appear. Such a method represents a necessary step toward high-dependability system engineering for numerous possible application domains, including distributed software systems, sensor networks, robot operation, complex scripts for spacecraft integration and testing, and autonomous systems. Currently, available tools and methods that start with a formal model of a system and mechanically produce a provably equivalent implementation are valuable but not sufficient. The "gap" that current tools and methods leave unfilled is that their formal models cannot be proven to be equivalent to the system requirements as originated by the customer. For the classes of systems whose behavior can be described as a finite (but significant) set of scenarios, we offer a method for mechanically transforming requirements (expressed in restricted natural language, or other appropriate graphical notations) into a provably equivalent formal model that can be used as the basis for code generation and other transformations.[5]

Naveen Alwandi, Manjunath BC, "Experiences with AUTOSAR compliant Autocode generation using TargetLink", Increased safety, comfort, and emission norms are pushing the complexity of vehicle systems up exponentially. Model-based development processes have increasingly been adopted for the development of automotive embedded control software to help implement complex systems and reduce the development time. Model-based and auto code technology has become mature and brings many advantages to automotive software development. In parallel, a consortium of major OEMs and suppliers are driving toward the standard specification of automotive software architecture, AUTOSAR (AUTomotive Open System Architecture). AUTOSAR would enable flexibility for product modification, upgrade and update scalability of solutions within and across product lines. To model algorithms and generate AUTOSAR compatible Autocode has become the necessity for projects using AUTOSAR architecture.[6]

Dominique Orban, "Templating and Automatic Code Generation for Performance with Python", Parameterizing source code for architecture-bound optimization is a common approach to high-performance programming but one that makes the programmer's task arduous and the resulting code difficult to maintain. Certain parameterizations, such as changing loop order, may require elaborate code instrumenting that distracts from the main objective. In this paper, we propose a templating and automatic code generation approach based on standard Python modules and the Opal library for algorithm optimization. Advantages of our approach include its programmatic simplicity and the flexibility offered by the templating engine. We provide a complete example for the matrix multiply where the optimization concerning blocking, loop unrolling, and compiler flags takes place [7]

Bran Selic, "Complete High-Performance Code Generation from UML Models" Automation is the traditional industrial means for improving productivity and product quality. We explore both the theoretical and pragmatic issues involved in automating the development of complex embedded software. In particular, we focus on techniques for fully automatic, complete, generation of high-performance code directly from high-level design models. "Full" code generation extends beyond mere code skeleton. In effect, it means directly programming, testing, and debugging complex applications using higher-level modeling constructs, such as those provided in UML.[8]

 DOI : 10.35291/2454-9150.2022.0141

Van Cam Pham, Ansgar Radermacher, Sébastien Gérard and Shuai Li, "Complete Code Generation from UML State Machine", An event-driven architecture is a useful way to design and implement complex systems. The UML State Machine and its visualizations are a powerful means of modeling the logical behavior of such an architecture. In Model-Driven Engineering, executable code can be automatically generated from state machines. However, existing generation approaches and tools from UML State Machines are still limited to simple cases, especially when considering concurrency and pseudo-states such as history, junction, and event types. This paper provides a pattern and tool for a complete and efficient code generation approach from UML State Machine. It extends IF-ELSE-SWITCH constructions of programming languages with concurrency support. The code generated with our approach has been executed with a set of state-machine examples that are part of a test- suite described in the recent OMG standard Precise Semantics of State Machine. [9]

Neil Audsley, Iain Bate, Steven Crook-Dawkins, "Dependable and ubiquitous Autocode Generation", “Automatic Code Generation” is a process of deriving programs directly from a design representation. Many commercial tools provide this capability. Whilst these tools provide greater flexibility and responsiveness in design, the market is technology-focused and immature. No infrastructure or established theory exists which could be used to deploy the technology across large projects whilst upholding coding standards and safety requirements. The objective of this paper is to develop a model or architecture for code generation that will be sufficiently well-defined and unambiguous to support formal reasoning whilst also retaining sufficient expressive power to be useful. These models are based on statically defined mappings. [10]

III. PROPOSED SYSTEM

This chapter includes a brief description of the proposed system and explores the different modules involved along with the various models through which this system is understood and represented. The Base Logic is that AutoCodeApp works based on a custom-made key-value Configuration based on REGEX. It possesses a declarator Front End, which is WebApp made using Flutter Framework that defines the user requirements inside standardized API specifications for (example OpenAPI 3.0, SwaggerAPI, Custom, etc.). A well-defined Spec file is generated using the inputs by the User and then Validated before sending to the code generator. Then Code Generator Engine curates a Specificized File which is parsed to generate an API based on the Configuration Key-Value pairs using Custom REGEX.

Our first module is the reactive frontend, here user interacts with Webapp and enters credentials, and proceeds to select their desired framework[3] other modules start working on the requirements received, builthe der engine works on the instructions generated based on the requirements and automates the process of providing Code-based of pre-made functions and templates according to user’s need. Henceforth, the template gets downloaded in zip format. The dashboard reflects all the ongoing activities performed by the user. In figure 3.1, we show the detailed system architecture.

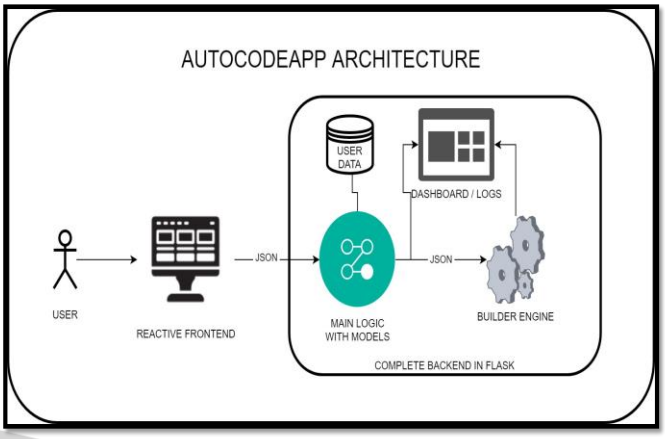

Figure 3.1. System Architecture

Details of Modules

AutoCodeApp tends to integrate various modules to facilitate the code generation process and increase efficiency.

The modules are:
1. Reactive Frontend
2. Logic with models
3. Dashboard
4. Builder engine

A.Reactive Frontend

Built to take advantage of modern web features, reactive web applications are a powerful way of developing web apps. Unlike server-side rendered apps that need expensive round-trips, the reactive web quickly reacts to any user interaction. The pressure on the backend servers is also decreased, making these apps scale much better under heavy loads, all leading to a smoother user experience[8]. The reactive web model ensures you don’t need highly skilled front-end developers and a multitude of code frameworks to create complex, data-rich interfaces that adapt to changes in real-time.

B.Logic with models[8]

There are three ways to add custom application logic to models:

Remote methods - REST endpoints mapped to Node functions.

Remote hooks - Logic that triggers when a remote method is executed (before or after).

 DOI : 10.35291/2454-9150.2022.0141

Operation hooks - Logic triggered when a model performs create, read, update, and delete operations against a data source.

C. Dashboard

A dashboard is a type of graphical user interface[9] that often provides at-a-glance views of key performance indicators (KPIs) relevant to a particular objective or business process.

In another usage, "dashboard" is another name for "progress report" or "report" and is considered a form of data visualization. The “dashboard” is often accessible by a web browser and is usually linked to regularly updating data sources[10]. Here, Dashboard reflects the overview and API utilization, and configuration.

D. Builder engine

As the name suggests, the builder engine works with user-required data and utilizes models, and helps in building templates[7].

Configuration based template driven code generator using blocky UI[9].

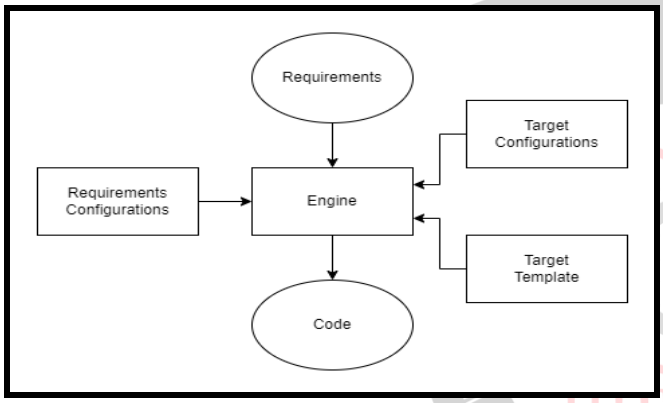

Figure 3.2. Core logic

It becomes a lot easier to maintain and upgrade if we divide entire code generator engine into 4 separate parts:

1.Requirement configuration
2.Target configuration
3.Target template
4.Main engine

In this architecture, we can reuse just one codegen engine for any application without making any changes to core, i.e. If we wish to generate code using any requirement configurations and port it to any desired target framework, we can do that by just adding new configs and templates while not making any changes in the existing code generation engine[4].

The main core will parse the requirements configurations and Target configurations based on which it will use target template to generate desired code[2].

For example, we generated API generator by providing OpenAPI3.0 configurations and Flask Configurations and template Flask app, Code generate API using Flask framework[5]. Now if we wish to make Express API using the same requirements configurations, then we won’t need to make any changes to the code generator. By just making congrats and template for Express we can generate code in that framework too. Next if we wish to add SwaggerAPI support then we can simply add it’s configurations and be ready to accept inputs in that format.

IV. Result and Analysis

Easily scalable code generator developed without having to modify and complicate code generator engine

Need to write configurations and templates separate for each language and framework

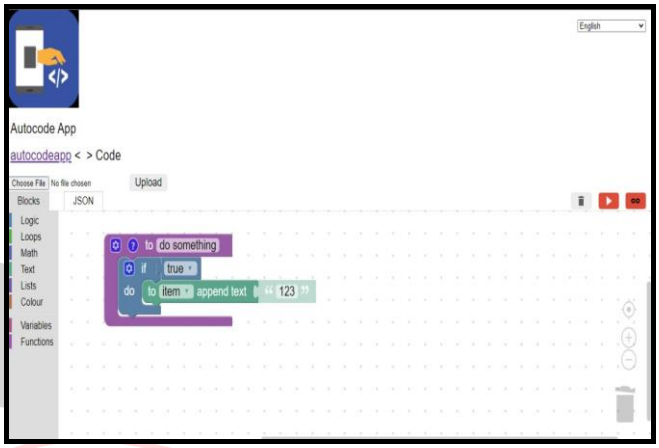

Figure 4.1. GUI

User after logging in using credentials sees the blocks. By clicking and dragging each block user can design the desired API. Words in the UI are generalized semi technical words for easy understanding by nontechnical background users. Where they can design features they wish in their API while generating it’s logic and the functions defining the logic.

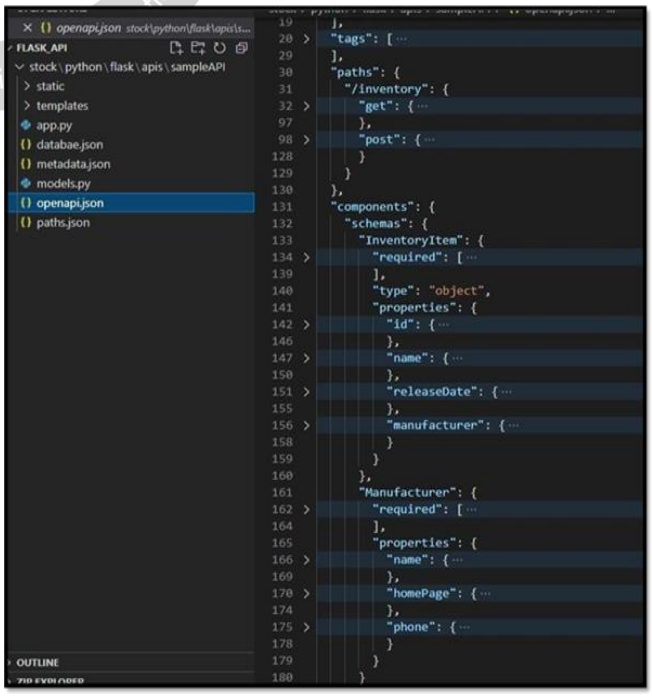

Figure 4.2. OpenAPI 3.0 specification in form of JSON

 DOI : 10.35291/2454-9150.2022.0141

Figure 4.2. shows a sample Manufacturing firm API specification written in OpenAPI 3.0 specification which will be generated using blocky UI[9] for code generation

```
# OpenAPI version
APISPEC = ["openapi"]

# INFO section
INFO = ["info"]
TITLE = ["info", "title"]
DESCRIPTION = ["info", "description"]
VERSION = ["info", "version"]
CONTACT = ["info", "contact"]
EMAIL = ["info", "contact", "email"]
LICENSE = ["info", "license"]
LICENSENAME = ["info", "license", "name"]
LICENSEURL = ["info", "license", "url"]

# SERVER section
SERVER = ["servers"]
SERVERSHOST = ["servers", "host"]
SERVERSPORT = ["servers", "port"]
SERVERSURL = ["servers", "url"]

# TAGS section
TAGS = ["tags"]
TAGS_X_NAME = ["tags", DEFAULTINDEX, "name"]
TAGS_X_DESCRIPTION = ["tags", DEFAULTINDEX, "description"]

# PATH TAGS section
GET_TAGS = ["paths", DEFAULTPATH, "get", "tags", DEFAULTINDEX]
POST_TAGS = ["paths", DEFAULTPATH, "post", "tags", DEFAULTINDEX]
PUT_TAGS = ["paths", DEFAULTPATH, "put", "tags", DEFAULTINDEX]
DELETE_TAGS = ["paths", DEFAULTPATH, "delete", "tags", DEFAULTINDEX]
UPDATE_TAGS = ["paths", DEFAULTPATH, "update", "tags", DEFAULTINDEX]

...
SCHEMA / DATABASE
...
# SCHEMES section
DATABASELIST = ["components", "schemas"]
```

Figure 4.3. OpenAPI 3.0 Configuration keys

Figure 4.3. shows Similar configurations for OpenAPI specifications are shown in figure 4.3 on basis of which the code engine will able to parse the input Specification file generated by blocky

```
def autocodeapp():
    return '<h1> AUTOCODEAPP WELCOMES YOU </h1>'
"""
FLASKSERVER = """
if __name__ == '__main__':
    app.run()
"""
FLASKROUTE = "@app.route('|route|', methods=['|method|'])\n"
FLASKDEF = "def |fname|(): \n"
FLASKCOMMENT = "    ''' |comment| ''' \n"
FLASKFORMREQUEST = "    |name| = request.form['|name|'] \n"
FLASKARGS = "  \n"
FLASKRETURN = "    return '|return|' \n"


'''
FLASKSQLALCHEMY TABLE TEMPLATE
'''

SQLALCHEMYINIT = """
from flask_sqlalchemy import SQLAlchemy
from app import db

"""
CLASSHEAD = "class |classname|(db.Model):\n"
CLASSBODY = "    |columnname| = db.Column(db.|datatype|, primary_key= |isprimary| , unique= |unique| , nullable= |nullable|) \n"
CLASSRELATED = "    |columnname| = db.relationship('|related|', lazy='select', backref=db.backref('|columnname|', lazy='joined')) \n"
CLASSINIT = "\n    def __init__(self|columnnamess|): \n"
CLASSINITBODY = "        self.|columnname| = |columnname| \n"
CLASSADD = """
    def addthis(self):
        db.session.add(self)
        db.session.commit()
        \n"""
```

Figure 4.4. Template for Flask-framework parsing

A template of a desired framework has to be generated for generating code in the same. Figure 4.3. shows Template for flask framework where pieces of code are written in the form of key value pairs which will be read by the code generation engine

Parameter	Existing System	Autocodeapp
working	Single complex Engine	Configuration based
Specification	Uses OpenAPI 2.0	Compatible with any possible specification
Interface view	Text editor	Click and drag UI

Figure 4.5. Comparison between existing and proposed system

V. CONCLUSION

In a world of diverse technologies and increased reliance on software systems the role of automated code generating tools is extremely important. We have introduced how it is intended to provide complete, efficient, and automated coding from pre-defined templates[7].

A common solution to this method may involve processing using a single complex engine, which will be more complex as we add a new framework and specific support to it. By using AutoCodeApp we can make that easier by setting up templates based on each support without adding complex elements to the main engine.

An existing system like Swagger uses the old OpenAPI2.0. Even if they add OpenAPI3.0 details they still need to convert the new data to their old standard which keeps the overlay engine in the middle class. This is no longer a problem, as all configurations are accessed by a central engine.

We argue that our tool produces functional code that runs faster at event processing speeds and is smaller in usable size[8]. It sets coding standards that can help analyze the situation and ensures that the next generation of AutoCode[5] tools are easy to think about and use as well as a manageable process of translating from ideas to code effectively. Often, the coding process seems long and tedious as any coding writer has to write every single line and memorize the rules of coding. However, with Block-based coding, this process becomes simpler and faster as the visual element also helps in making the process smoother and more fun when it comes to planning long projects. Since these coding languages were developed by engineers for use by other engineers, many people who do not have formal training can now have a way to learn to write code in less time, so that they can work on smaller programs. Blockchain-based coding is a solution, as it is accurate and does not require formal training to master it.

 DOI : 10.35291/2454-9150.2022.0141

It's easy and any beginner can take it after a few days. Ultimately this approach will help us meet the minimum requirements for default coding for high-level non-programmers[1].

ACKNOWLEDGMENT

We take this opportunity to express our deep sense of gratitude to our project guide and project coordinator, Mrs. Poonam Thakre, for her continuous guidance and encouragement throughout our Project work. It is because of her experience and wonderful knowledge, we can fulfill the requirement of completing the project within the stipulated time. We would also like to thank Dr. Jitendra Saturwar, Head of the computer engineering department for his encouragement, whole-hearted cooperation, and support.

We would also like to thank our Principal, Dr. J. B. Patil, and the management of Universal College of Engineering, Vasai, Mumbai for providing us with all the facilities and a work-friendly environment. We acknowledge with thanks, the assistance provided by departmental staff, library, and lab attendants.

REFERENCES

[1] Jeffrey Smith, Mieczyslaw Kokar, Kenneth Baclawski, "Formal Verification of UML Diagrams: A First Step Towards Code Generation", ResearchGate, 2001.

[2] Gergely Pintér, István Majzik, "Automatic code generation based on formally analyzed UML statechart models", ResearchGate, 2003.

[3] Ivo Damyanov, Nick Holmes, "Metadata Driven Code Generation Using .NET Framework", International Conference on Computer Systems and Technologies, 2004.

[4] Ingo Stürmer, Mirko Conrad, "Code Generator Certification: A Test Suite-oriented Approach", ResearchGate, 2004.

[5] Michael G. Hinchey, James Rash, and Christopher A. Rouff, “Towards a Fully Formal Approach to Automatic Code Generation”, NASA, 2005.

[6] Naveen Alwandi, Manjunath BC, "Experiences with AUTOSAR compliant Autocode generation using TargetLink", dSPACE User Conference, 2010.

[7] Dominique Orban, "Templating and Automatic Code Generation for Performance with Python", ResearchGate, 2013.

[8] Bran Selic, "Complete High-Performance Code Generation from UML Models", ResearchGate, 2014.

[9] Van Cam Pham, Ansgar Radermacher, Sébastien Gérard and Shuai Li, "Complete Code Generation from UML State Machine", 5th International Conference on Model- Driven Engineering and Software Development, 2017.

[10] Neil Audsley, Iain Bate, Steven Crook-Dawkins, "Dependable and ubiquitous Auto code Generation", University of York, 2018.

 DOI : 10.35291/2454-9150.2022.0141