Abdo Talab ALGhazali

PhD Candidate

Department of Computer Engineering and Automation,

Faculty of Mechanical and Electrical Engineering,

Damascus University,

Damascus, Syria.

E-Mail: abdo.ghazali2026@damascusuniversity.edu.sy

An Expert System to Generate Domain-Specific Programs using Natural Language Processing

Primary Supervisor: Prof. Sameer Kharaman

E-Mail: kharaman@damascusuniversity.edu.sy

Co-Supervisor: Dr. Mohammed Mazen AlMahayri

E-Mail: Mazen.almahayri@damascusuniversity.edu.sy

Funding: This research received no external funding

Abstract

The rapid advancement of artificial intelligence and natural language processing (NLP) has significantly influenced software development practices. One promising direction in this domain is generating the code automatically from the description of natural language. This research proposes a domain-specific expert system that generates source code automatically from user requirements written in natural language. Unlike general-purpose code generating systems, the proposed expert system focuses on predefined program domains such as student grading system. The expert system processes textual input using NLP algorithms, extracts program elements, converts them into an intermediate representation, and generates executable code. Experimental evaluation demonstrates that restricting the domain significantly improves the accuracy and reliability of generated programs.

1. Introduction

Software development traditionally requires programmers to translate user requirements into programming languages. This process is often time-consuming and prone to misunderstandings between users and developers.

Recent developments in AI-based tools such GitHub Copilot, ChatGPT have demonstrated the potential of automated code generating using natural language descriptions.

Despite these advances, most existing systems generate code for general programming tasks and lack domain constrains. This often leads to inaccurate or inconsistent code outputs.

To address this limitation, this research proposes an expert system for generating a domain-specific code. In this expert system, the type of program is defined in advance (e.g., student grading system). User requirements are then expressed in natural language and processed using NLP algorithms to generate the corresponding source code.

This domain restriction allows the system interpret the intention of the user in a better way and generate a code more reliably.

2. Research Problem

Automatic code generation from natural language remains a challenge due to several factors:

- Ambiguity of natural language.
- Multiple ways to express the same requirement.
- Lack of domain constraints in general code generation systems.
- Difficulty in translating natural language semantics into executable program logic.

Therefore, the main research question is:

How can natural language processing algorithms be used to generate accurate program code for predefined software domains?

3. Research Objectives

This research aims at:

- ❖ Designing an expert system that converts natural language requirements into an executable code.
- ❖ Developing a domain-specific code generation models.
- ❖ Implementing the expert system for predefined applications such as student grading system.
- ❖ Evaluate the accuracy and effectiveness of the generated code.

4. Related Work

Code generation research has evolved through several approaches:

- ❖ Rule-Based Code Generation

Early systems relied on predefined grammar rules that mapped structured input into programming constructs. Despite their accuracy, these systems lacked flexibility.

- Model-Driven Engineering

Model-driven approaches automatically convert system models into source code. However, they still require structured modeling languages.

- Neural Code Generation

Recent research uses deep learning models trained on large datasets of code and natural language descriptions. For example, AI-based assistants like GitHub Copilot generate code suggestions during development.

However, these systems are general-purpose and may produce incorrect code when the context is unclear. This research instead, focuses on domain-specific generation, which limits the scope, eliminates ambiguity and improves precision.

5. Proposed Methodology

The proposed expert system consists of several stages:

5.1. Domain Definition

The first step is defining the application domain. In this research, many domains are selected, such as:

- Student grading system (result, degree, etc.)

Each domain includes predefined program structures and operations.

5.2. Natural Language input

Users describe program requirements using natural language, for example:

- “Conclude the result of a student from the score.”
- “Conclude the degree of a student from the scores.”

5.3. Natural Language Processing

The system processes the input text using NLP techniques that include:

- Tokenization
- Part-of-speech- tagging

- Dependency parsing

These processes identify:

- Inputs
- Variables
- Operations
- Conditions
- Outputs

5.4. Intermediate Representation

The extracted information is transformed into an intermediate representation (like pseudo) that describes the logic of the program, for example:

Program: Student Grade System

Input: mark

Operation: if

Condition: mark>=60

Output: result (passed, failed)

This representation acts as a bridge between natural language and source code.

5.5. Code Generation

The code generator converts the intermediate representation into source code into a target programming language.

6. System Architecture

The architecture of the proposed system consists of the following modules, as shown in Figure 6-1.

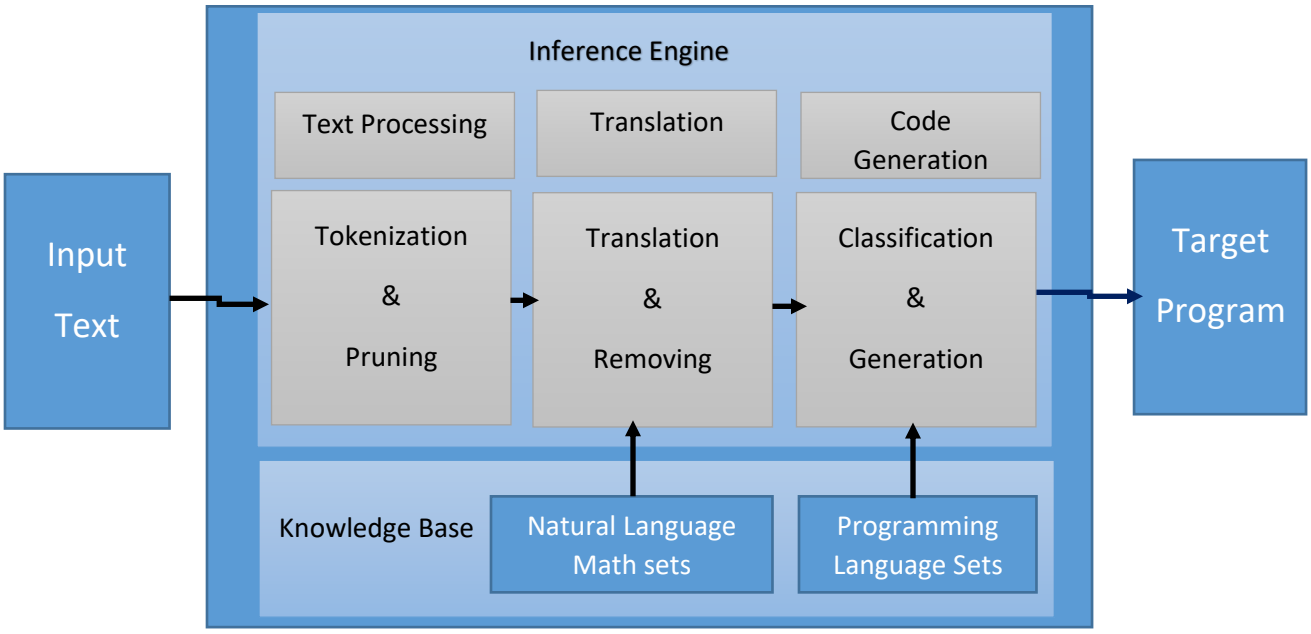

Figure. 1

Overall Architecture of the proposed expert system

The expert system is designed using the Visual C# programming language. The expert system goes through several stages:

Stage 1: (Program Execution)

After running the program that represents the expert system, the user interface appears. Then we select the natural language and programming language, as well as the program to be generated; for example: English-Python (Student Grade Program)

Stage 2: (Text Loading)

- Task selection: the work is divided into several tasks, from which we select the required task.

Stage 3: (Text Processing)

- Text loading and tokenization: after loading the text, it is converted into tokens stored in mathematical set in the form of symbols and words.
- Removal of redundancies, such as non-visible symbols (line breaks)

Stage 4: (Translation)

- Translation into the intermediate language: in this stage, translation is performed from the natural language into an intermediate language that includes programming language vocabulary, based on a knowledge base consisting of mathematical sets describing both the programming language and the natural language.
- Removal of repeated words.

Stage 5: (Code Generation)

- Token classification: tokens are classified to facilitate the code generation process, such as keywords, inputs, outputs, conditions, segments, etc.
- Code generation: the code is generated based on the output of the classification process and according to the structure of the programming language described through the mathematical sets in the knowledge base. Generation is performed gradually; symbol by symbol and word by word, not sentence translation.

Stage 6: (Target Program Generation)

- Generating the start of the target program
- Generating the first task.
- Generating the second task.
- Generating the end of the target program.

The execution steps are presented vertically as follows:

Selecting the natural language (Arabic or English)

↓

Selecting the programming language (Python or C++)

↓

Selecting the program

↓

Selecting the task

↓

User Input (Natural Language)

↓

Natural Language Processing

↓

Semantic Analysis

↓

Intermediate Representation

↓

Code Generation Engine

↓

Generated Program

Each module plays a role in converting natural language instructions into executable code.

Knowledge Base:

The knowledge base consists of mathematical sets. Some of them are used to describe the natural language; therefore, the system can be extended to write in different languages by developing different mathematical sets. For example, mathematical sets of describing the Arabic language, and other ones for describing the English language.

Other mathematical sets are used to describe the required programming language; therefore, the system can be extended to write in different programming languages by developing different sets. For example, mathematical sets for describing the Python language, and other ones for describing the C++ programming language.

The knowledge base consists of mathematical sets used to describe natural languages and to describe programming language instructions. For example, the set of “if” in Python language:

```
// Python
string[] ifonlyPython = { "if", "(", "condition", ")", ":", "statements", };
string[] ifelsePython = { "if", "(", "condition", ")", ":", "statements1",
"else", ":", "statements2" };
```

Program generation is performed according to this description using an inference method, symbol by symbol and word by word, without any generation errors, because the inference engine follows the exact structure of the instruction without any addition or omission.

As for sentences such as conditions, a dedicated function is called to generate the condition. Accordingly, generation is performed using a hyper-generation approach, so that the sentence is fully structured. A fully structured sentence is one that contains all elements of the set without any deficiency.

Generation is performed in a manner similar to assembly (rastering), using symbols, letters, and words according to the exact structure of the programming language instructions (not as full sentences).

When the programming language is changed, other mathematical sets are used. For example, the “if” set in C++ language:

```
// C++

string[] ifonly = { "if", "(", "condition", ")","{", "statements","}" };
```

```
string[] ifelse = { "if", "(", "condition", ")", "{","statements1","}", "else",
"{","statements2", "}" };
```

Input Error Correction

The system has been improved to correct user input errors using exceptions. If the user makes an input error, the system displays an error message. The system then gives the user another opportunity to regenerate suggesting the original input text to the user.

Artificial Intelligence Models:

Artificial Intelligence Models to be developed for *Student Grade Program* are:

- Arabic-Python Model
- English-Python Model
- Arabic-C++ Model
- English-C++ Model

Detailed Explanation

- Program execution: The Expert System is executed.
- Selection of program, natural language and programming language. For Example: Task (Student result: Passed or Failed).
- Loading text into a list (mathematical set): The input text is loaded into a list represented as a mathematical set.
- Tokenization of text (word decomposition): The text is converted into a set of tokens (words). If no text is found, the system displays the message “No Text Available”.
- Removal of special symbols (cleaning process): Removal of line breaks and special characters.
- Translation Process: Before starting the translation process, the knowledge base must be built. The knowledge base consists of mathematical lists used to describe natural languages. Examples:

  Description of the “task type” in English as a mathematical list:

  ```
  string[] taskTypeE = {"slice","slice", "slic","slice","Slice","slice" };
  ```

Description of Python programming language symbols as a mathematical list:

```
string[] PythonSymbols = { "if", "else", "(", ")", ":" };
```

Description of C++ programming language symbols as a mathematical list:

```
string[] symbols = {"if", "else", "(", ")", ";","{","}",":" };
```

Description of degrees in English language as a mathematical list:

```
string[] degreesE = {"honor", "excellent", "very good", "good", "acceptable", "failed" };
```

Translation process:

- Clear the translation list content
- Translation to intermediate language
- For each word in the processed word list:

If the keyword list contains the world (from the processed text), then:

Add the corresponding intermediate translation to the compilation list (lstcompile).

Processed words list → translation list.

If the intermediate list contains the word (from processed words):

Add the next word from the intermediate list (its translated equivalent) to the translation list.

- **Outputs:**

If the output list contains the word (from processed words):

Add the corresponding translated word (intermediate representation) to the compilation list (lstcompile).

- **Classification:**

Classification of words from the translation list into new lists.

Translation list → intermediate language

Classification result:

- ✓ Keywords list
- ✓ Input (data) list
- ✓ Output list
- ✓ Condition list
- ✓ Order list
- ✓ Domain list
- ✓ Task list

Intermediate list

Classification of words from the translation list into new lists:

Clear the contents of all previous classification lists.

Keyword classification:

For each word in the translation list (lstcompile2):

Classify words according to membership rules in mathematical sets.

Condition classification: Classify condition-related words.

And, similarly, for the rest classification lists

- **Code Generation**

To generate the code, we first perform the following:

Clear all elements (contents) of the code list (lstcode)

// C#

lstcode.Items.Clear();

Generation of program components:

Start generation:

For task1 (student result task: passed or failed)

```
// C#
if (taskNoGeneration==1)
{
```

```
        begingPythonenerate();
}
```

The assembly process starts by calling the function *begingPythonenerate()* to generate the beginning of the program.

Keyword generation:

The function *ifelsePythonGenerate()* is used. The code is generated (word by word or symbol by symbol), according to the structure of the if-else statement in Python. This structure has been fully modeled using mathematical sets; therefore, generation is performed without any errors. For example:

```
// C#

public void ifelsePythonGenerate()
      {
          string sifelse = "";
          bool blogical = false;
          bool bstatements1 = false;
          bool bstatements2 = false;
          int ifelsecount = ifelsePython.Count();
          for (int j = 0; j < ifelsecount; j++)
          {
              if (symbols.Contains(ifelsePython[j]))
              {
                  sifelse = sifelse + ifelsePython[j];
              }

              // logical test
              if (ifelsePython[j] == "condition")
              {
                  string logical_test;
                  logical_test = logicalGenerate();
                  sifelse = sifelse + logical_test;
                  blogical = true;
              }

              // statements1
              if (ifelsePython[j] == "statements1")
              {
                  string text1;
                  text1 = statements1PythonGenerate();
                  sifelse = sifelse + text1;
```

```
            text1 = Environment.NewLine;
            sifelse = sifelse + text1;
            bstatements1 = true;
        }
        // statements2
        if (ifelsePython[jj] == "statements2")
        {
            string text1;
            text1 = statements2PythonGenerate();
            sifelse = sifelse + text1;
            bstatements2 = true;
        }
    }//end of for
    lstcode.Items.Add(sifelse);

} // end of if else Python  generate
```

Similarly, for the rest of the generation operations.

- Target Program Generation

At the end of the code generation process, all components are combined into a single program. Then the program is tested by using Python via Visual Studio, or using the C++ language via Visual C++ 6.0.

7. Experimental Evaluation

To evaluate the proposed system, experiments are conducted using a dataset of natural language requirements for predefined applications. Evaluation metrics include:

- Accuracy of generated code
- Execution correctness
- Generation time
- Code readability
- Syntactic Correctness

7.1. Experiment 1: in 12/3/2026 – 3 pm

// generated code in Python

```
import os
def clear_screen():
 if os.name == 'nt':
   _ = os.system('cls')
 else:
   _ = os.system('clear')
clear_screen()
grade=70
grade=input("Enter a student grade, grade=");
grade=int(grade)
if(grade>=60):print ( "succeeded")
else:print ( "failed")
```

According to international metric, the following tables have been done:

Metric	ChatGPT-5	GitHub Copilot	Generated Code
Blue	0.80 – 0.90	0.78 – 0.88	0.95
Code Blue	0.85 – 0.9	0.80 – 0.88	0.90
Exact Match/Functional	90% - 100%	85% - 95%	100%
Syntactic Correctness	Correct	Correct	Correct
Readability	High	High	High

Quality Tier Classification:

Range	Classification
0.90 – 1.00	State-of-the-art
0.80 – 0.89	High Quality
0.70 – 0.79	Medium
<0.70	Low

Score = (0.95 + 0.90 + 1.00 + 1.00 + 1.00)/5

Result: 0.97 = State-of-the-art

7.2. Experiment 2: in 12/3/2026 – 3 pm

```
// generated code in C++
#include <iostream>
using namespace std;
int main()
{
int grade=70;
cout << "Enter a student grade (0-100) = ";
cin >> grade;
if(grade>=60)
        {cout<< "succeeded\n";}
else
        {cout<< "failed\n";}
return 0;
} // end of main
```

Metric	ChatGPT-5	GitHub Copilot	Generated Code
Blue	0.80 – 0.90	0.78 – 0.88	0.95
Code Blue	0.85 – 0.9	0.80 – 0.88	0.90
Exact Match/Functional	90% - 100%	85% - 95%	100%
Syntactic Correctness	Correct	Correct	Correct
Readability	High	High	High

Score = (0.95 + 0.90 + 1.00 + 1.00 + 1.00)/5

Result: 0.97 = State-of-the-art

7.3. Experiment 3: in 16/3/2026 – 7:56 pm

// generated code in Python

```
import os
def clear_screen():
 if os.name == 'nt':
   _ = os.system('cls')
 else:
   _ = os.system('clear')
clear_screen()
grade=70
grade=input("Enter a student grade, grade=");
grade=int(grade)
if    95 <= grade <= 100: print( " honor " )
elif 85 <= grade <= 94 : print( " excellent " )
elif 75 <= grade <= 84 : print( " very good " )
elif 65 <= grade <= 74 : print( " good " )
elif 60 <= grade <= 64 : print( " acceptable " )
else: print( " failed " )
```

Metric	ChatGPT-5	GitHub Copilot	Generated Code
Blue	0.80 – 0.90	0.78 – 0.88	0.90 – 0.95
Code Blue	0.85 – 0.9	0.80 – 0.88	0.88 – 0.90
Exact Match/Functional	90% - 100%	85% - 95%	100%
Syntactic Correctness	Correct	Correct	Correct
Readability	High	High	High

Score = (0.92 + 0.89 + 1.00 + 1.00 + 1.00)/5

Result: 0.96 = State-of-the-art

7.4. Experiment 4: in 14/4/2026 – 3pm

// Average: generated program in C++

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
string name="";
int number_of_marks=12;
cout<<"input student name: ";
cin>>name;
cout<<"input student marks: ";
cout<<"input number of marks: ";
cin>>number_of_marks;
int marks[100];
int size=number_of_marks;
for (int i=0; i< size;i++){cout<<"Enter a new mark: "; cin>>marks[i];}
int sum=0;
for (int i=0; i< size;i++){sum = sum + marks[i];}
cout<<endl;
cout<<"sum = "<<sum;
double average=(double)sum/number_of_marks;
cout<<endl;
cout<<"Average= "<<average <<endl;
cout<<endl;
cout<<endl;
return 0;
```

}

// Executed by Dev-C++ 5.11

Compilation results...

- Errors: 0

- Warnings: 0

- Output Filename: E:\2026\ph2026\Dev\Sum\CSum.exe

- Output Size: 1.83608627319336 MiB

- Compilation Time: 1.20s

input student name: Abdo

input student marks: input number of marks: 5

Enter a new mark: 100

Enter a new mark: 90

Enter a new mark: 95

Enter a new mark: 85

Enter a new mark: 75

sum = 445

Average= 89

Process exited after 20.8 seconds with return value 0

Press any key to continue . . .

Metric	ChatGPT-5	GitHub Copilot	Code
Blue	0.88	0.78	0.40

CodeBlue	0.90	0.82	0.50
Exact Match/Functional	92%	85%	60%
Syntax	1.00	1.00	0.30
Readability	0.90	0.85	0.50

Score = (0.40 + 0.50 + 0.60 + 0.30 + 0.50)/5

Result: 0.46 = Low

7.5. Experiment 5: in 14/4/2026 – 3 pm

// Average: corrected generated program in C++ using “vector”

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
string name="";
int number_of_marks=12;
cout<<"Enter student name: ";
cin>>name;
cout<<"input number of marks: ";
cin>>number_of_marks;
if(number_of_marks<=0 || number_of_marks>100)
{
        cout<<"Invalid number of marks!"<<endl;
        return 1;
}
```

```
vector<int>
marks(number_of_marks);
for (int i=0; i< number_of_marks;i++)
{
        cout<<"Enter mark "<<i+1<<": ";
        cin>>marks[i];
}
int sum=0;
for (int i=0; i< number_of_marks;i++)
{sum+= marks[i];}
double average=(double)sum/number_of_marks;
cout<<endl;
cout<<"\n Total= "<<sum<<endl;
cout<<"Average= "<<average <<endl;
cout<<endl;
return 0;
}
```

// Executed by Dev-C++ 5.11

Compilation results...

- Errors: 0
- Warnings: 0
- Output Filename: E:\2026\ph2026\Dev\test1\average.exe
- Output Size: 1.84862899780273 MB
- Compilation Time: 1.20s

Enter student name: Ali

input number of marks: 5

Enter mark 1: 90

Enter mark 2: 100

Enter mark 3: 90

Enter mark 4: 80

Enter mark 5: 80

Total= 440

Average= 88

Process exited after 10.04 seconds with return value 0

Metric	ChatGPT-5	GitHub Copilot	Corrected Code
Blue	0.88	0.78	0.75
CodeBlue	0.90	0.82	0.85
Exact Match/Functional	92%	85%	85%
Syntax	1.00	1.00	1.00
Readability	0.90	0.85	0.80

Score = (0.75 + 0.85 + 0.85 + 1.00 + 0.80)/5

Result: 0.85 = High Quality

7.6. Experiment 6: in 14/4/2026 – 3pm

Gap Analysis

Metric	Difference from ChatGPT
Blue	-0.13
Code Blue	-0.05

Exact Match/Functional	-0.07
Readability	-0.10

7.7. Experiment 7: in 14/4/2026 – 3pm

Compare before and after correction

Metric	Code	Corrected Code
Blue	4/10	7.5/10
CodeBlue	0.5	0.85
Exact Match/Functional	6/10	8.5/10
Syntax	3/10	10/10
Readability	5/10	8/10

Preliminary experiments indicate that restricting the system to a specific domain significantly improves generation accuracy.

7.8. Experiment 8: in 14/4/2026 – 3 pm

To enhance the statistical validity and reliability of the evaluation, the experimental study was extended to include a total of 70 test cases representing a diverse range of input

Student ID	Size	Sum	Average
Student 1	5	400	80.0
Student 2	4	270	67.5
Student 3	6	522	87.0
Student 4	3	165	55.0
Student 5	7	674	96.29
Student 6	5	410	82.0
Student 7	8	620	77.5
Student 8	6	540	90.0
Student 9	4	300	75.0
Student 10	9	810	90

Summary of the experimental results for 70 samples

Metric	Value

Total Samples	70
Correct Executions	70
Accuracy	1.00
Mean Average	79.8
Std Deviation	10.2
Min Average	55.0
Max Average	96.5

8. Discussion

The results that domain-specific code generation is more reliable than general-purpose code generation. By limiting the program type, the system can better interpret user requirements and map them to programming constructs. However, several challengers remain:

- Handling complex program logic.
- Supporting multiple programming languages.
- Supporting multiple Natural languages.

9. Conclusion

This research presented an expert system for generation program code from natural language within predefined application domains. The proposed system uses natural language processing algorithms to analyze user requirements converting them into an intermediate representation, and generating executable code automatically.

Experimental results demonstrate that domain-specific restrictions improve the accuracy and reliability of generated programs. The expert system can help simplify software development and enable non-programmers to generate basic applications using natural language descriptions.

10. Future Work

Future research directions include:

- ❖ Extending the expert system to support additional software domains.
- ❖ Supporting multiple natural languages.
- ❖ Improving the multi-task program generation process and enhancing performance quality.

11. References

- Giorgio. M (2024). Secure Code Generation system by using Artificial intelligence, news.njit.edu.USA.
- (2024). Software Test Case Generation Using Natural Language Processing, Artificial Intelligence Evolution, ojs.wiserpub.com.
- (2023). Arabic NLP 2023, Association for Computational Linguistics (ACL), Stroudsburg, USA.
- (2023). NLP-OSS (Open Source Software) 2023, Association for Computational Linguistics (ACL), Stroudsburg, USA.
- (2023). Natural Language Processing in ChatGPT, Open Artificial Intelligence Company (open AI), Jenni. Ai/ar/chat-gpt/nlp.
- E. Dehaerne et al (2022). Code Generation Using Machine Learning, IEEE Access.
- (2020). "Cognition". Lexico. Oxford University Press and Dictionary.com.

12. Author Names

- Abdo Talab ALGhazali

PhD Candidate

Department of Computer Engineering and Automation,

Faculty of Mechanical and Electrical Engineering,

Damascus University,

Damascus, Syria.

E-Mail: abdo.ghazali2026@damascusuniversity.edu.sy

- Primary Supervisor: Prof. Sameer Kharaman

E-Mail: kharaman@damascusuniversity.edu.sy

- Co-Supervisor: Dr. Mohammed Mazen AlMahayri

E-Mail: Mazen.almahayri@damascusuniversity.edu.sy

Funding: This research received no external funding