Ministry of Education and Science of Ukraine
Black Sea Universities Network

ODESA NATIONAL UNIVERSITY OF TECHNOLOGY

International Competition of Student Scientific Works

BLACK SEA SCIENCE 2022

PROCEEDINGS

ODESA, ONUT 2022

Ministry of Education and Science of Ukraine

Black Sea Universities Network

Odesa National University of Technology

International Competition of Student Scientific Works

BLACK SEA SCIENCE 2022

Proceedings

Odesa, ONUT 2022

UDC 001.8(063)

Black Sea Science 2022: Proceedings of the International Competition of Student Scientific Works / Odesa National University of Technology; B. Iegorov, M. Mardar (editors-in-chief) [*et al.*]. – Odesa: ONUT, 2022. – 749 p.

Proceedings of International Competition of Student Scientific Works «Black Sea Science 2022» contain the works of winners of the competition.

The author of the work is responsible for the accuracy of the information.

Organizing committee:

Prof. Bogdan Iegorov, D.Sc., Rector of Odesa National University of Technology, Head of the Committee

Prof. Maryna Mardar, D.Sc., Vice-Rector for Scientific and Pedagogical Work and International Relations of Odesa National University of Technology, Deputy Head of the Committee

Prof. Baurzhan Nurakhmetov, D.Sc., First Vice-Rector of Almaty Technological University (Kazakhstan)

Prof. Michael Zinigrad, D.Sc., Rector of Ariel University (Israel)

Prof. Plamen Kangalov, Ph.D., Vice-Rector for Academic Affairs of "Angel Kanchev" University of Ruse (Bulgaria)

Prof. Heinz Leuenberger, Ph.D., Professor of the Institute of Ecopreneurship of University of Applied Sciences and Arts (Switzerland)

Prof. Edward Pospiech, Dr. habil., Professor of the Institute of Meat Technology of Poznan University of Life Sciences (Poland)

Prof. Lali Elanidze, Ph.D., Professor of the Faculty of Agrarian Sciences of Iakob Gogebashvili Telavi State University (Georgia)

Dr. Dan-Marius Voicilas, Ph.D., Associate Professor of the Institute of Agrarian Economics of Romanian Academy (Romania)

Prof. Stefan Dragoev, D.Sc., Vice-Rector for Scientific Work and Business Partnerships of University of Food Technologies (Bulgaria)

Prof. Jacek Wrobel, Dr. habil., Rector of West Pomeranian University of Technology (Poland)

Dr. Mei Lehe, Ph.D., Vice-President of Ningbo Institute of Technology, Zhejiang University (China)

Dr. V. Kozhevnikova, Ph.D., Associate Professor of the Department of Hotel and Catering Business of Odesa National University of Technology, Secretary of the Committee

INTRODUCTION

International Competition of Student Scientific Works “Black Sea Science” has been held annually since 2018 at the initiative of Odesa National University of Technology (formerly Odesa National Academy of Food Technologies) with the support of the Ministry of Education and Science of Ukraine. It has been supported by Black Sea Universities Network (the Association of 110 higher education institutions from 12 countries of the Black Sea Region) since 2019, and by Iseki-FOOD Association (European Integrating Food Science and Engineering Knowledge into the Food Chain Association) since 2020.

The goal of the competition is to expand international relations and attract students to research activities. It is held in the following fields:

- Food science and technologies
- Economics and administration
- Information technologies, automation and robotics
- Power engineering and energy efficiency
- Ecology and environmental protection

The jury includes both Ukrainian and foreign scientists. In the 4 years that the competition has been held, the jury included scientists from universities of 24 countries: Angola, Azerbaijan, Benin, Bulgaria, China, Czech Republic, France, Georgia, Germany, Greece, Israel, Italy, Kazakhstan, Latvia, Lithuania, Moldova, Pakistan, Poland, Romania, Serbia, Slovakia, Switzerland, Turkey, USA.

At the same time, every year the geography has expanded and the number of foreign jury members has increased: from 46 jury members representing 25 universities from 12 countries in 2018, to 73 jury members of the 46 universities from 19 countries in 2022.

More than a thousand student research papers have been submitted to the competition from both Ukrainian and foreign institutions from 25 countries: China, Poland, Mexico, USA, France, Greece, Germany, Canada, Costa Rica, Brazil, India, Pakistan, Israel, Macedonia, Lithuania, Latvia, Slovakia, Romania, Kyrgyzstan, Kazakhstan, Bulgaria, Moldova, Georgia, Turkey, Serbia.

The interest of foreign students in the competition grew every year. In 2018, the students representing 15 institutions from 7 countries have submitted 33 works. In 2021 the number of submitted works increased to 73, authored by the students of 40 institutions from 18 countries.

The competition is held in two stages. In the first stage, student research papers are reviewed by members of the jury who are experts in the relevant fields. In the second stage of the competition, the winners of the first stage have the opportunity to present their work to a wide audience in person or online.

All participants of the competition and their scientific supervisors are awarded appropriate certificates, and the scientific works of the winners are included in the electronic proceedings of the competition. Every year the competition receives a large number of positive responses from Ukrainian and foreign colleagues with the desire to participate in the coming years.

3. INFORMATION TECHNOLOGIES, AUTOMATION AND ROBOTICS

A COMPILER OF DOMAIN-SPECIFIC LANGUAGE FOR "SMART-HOME" APPLICATIONS: DESIGN PRINCIPLES AND IMPLEMENTATION ISSUES

Author: Oleksandr Nelipa
Advisor: Mykola Tkachuk
V. N. Karazin Kharkiv National University (Ukraine)

Abstract. *The actuality to use of a domain-specific language (DSL) concept in such complex problem areas as the Internet of Things systems and "Smart-Home applications (SHA)" is motivated. The overview of the main methods and software tools for DSL design and implementation is done, and one possible scheme for their classifications is proposed. The approach to DSL compiler designing for SHA is proposed which is based on a configurable grammar rules system. All main functional blocks for the proposed DSL compiler are developed using such programming tools as Python and C++, and the first testing results of this implementation are obtained and analyzed. The effectiveness assessment for this compiler prototype is provided in the way to calculate of two quantitative metrics, and this one allowed to get the approximated weighted efficiency value of the compiler's usage about 16.75%. It shows the acceptable quality of the elaborated DSL compiler's prototype, allows to make the positive conclusions about the proposed approach, and to formulate some further work to be done in this research.*

Keywords: *compiler, domain-specific language, IoT, smart-home, design, software, effectiveness, metric*

I. INTRODUCTION

An efficient software development in such modern and high-tech application domains as the Internet of Things (IoT) system, and especially, for 'Smart-Home' applications (SHA) [1-2], supposes the usage of such new sophisticated and advanced design methods as a domain-driven development, a model-driven architecting, and some knowledge-based software tools and technologies [3].

SHA systems have some specific features and options which influence the appropriate problems for their design and implementation [2], see the Figure 1.

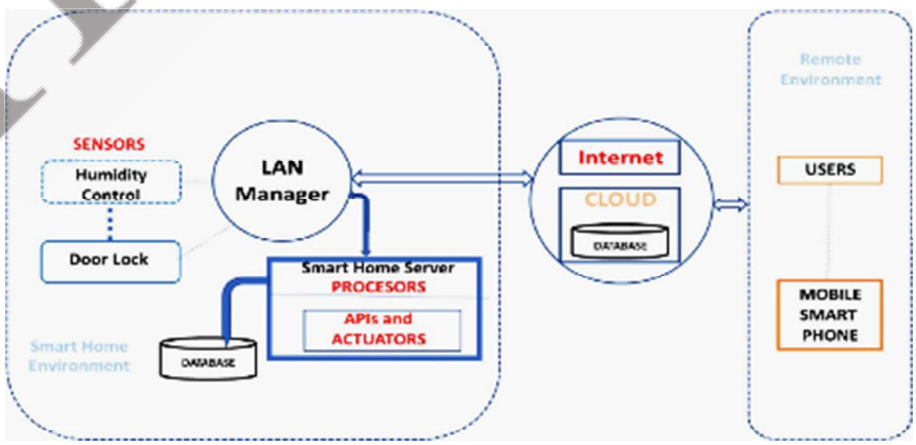

Fig. 1 – A reference conceptual architecture of IoT / SHA [2]

Some SHS specific features are:

- includes a lot of interconnected hardware and software components;
- any such a component has a lot of specific features and parameters;
- all components operate in real-time mode;
- operation environment is dynamic and changeable.

According to these issues the following main challenge for SHA-developers are facing with: how to design and to implement configurable and adaptable complex software and hardware solutions, taking into account different user's needs and requirements?

One of the recognized ways to achieve this aim is the usage of concepts and technologies of domain-specific languages (DSL) [4], which have to be created for a given application area, that finally allows to provide more effective and cheaper programming techniques in system development. One very important reason to use DSL is an opportunity to support variability and adaptivity of appropriate software solutions due to the elaboration of flexible grammar rules and building of correct domain dictionary (or a set of tokens).

The purpose of this study is to analyze some existing models, tools, design principles for DSL construction, to propose an approach to create DSL compiler for adaptive software development in SH applications, to provide compiler components prototyping and testing, and to get first results of effectiveness estimation for the proposed approach in the subject area 'Smart-Home' application.

II. MAIN DESIGN PRINCIPLES FOR SHA / DSL

2.1. Basic design approaches for DSL

There are 2 main types of domain-specific languages: external DSL and internal DSL (or also known as embedded DSL) [3]. External DSLs have their own syntax, which is, in most cases, separated from the application programming language. On the other hand, all internal DSLs use some general purpose language (GPL), but in fact, they just expend a specific subset of the functionalities of this language. One of the most important problems in the creation and future use of DSL is the availability of special language workbench (tools). These tools can be known as the specialized integrated development environments (IDEs) for defining, designing and creating DSL for specific needs.

The process of creating external DSLs consist of three key steps:

1. Definition of the semantic model;
2. Definition of the syntactic mode (abstract and concrete syntax);
3. Definition of rules of transformation (in other words, how abstract is translated to actual).

To determine the specific syntax of the language and to create the specific transformation rules by building a language translator there are some ready-made tools, which can be helpful. For example, ANTLR [5] allows generating lexical and syntax parser, language translator. To determine the semantic model of language, which describes a particular aspect of the system there are no special tools. To do so, every

DSL developer must independently describe the metamodel of the language by using GPL or, maybe, other DSL.

When creating an internal DSL, the most straightforward way is to select one of the GPL as a base (e.g. Java, Kotlin, C# etc.) and create a special library, based on the grammar of the chosen language. This custom library then could be used in a certain style, usually to manage particular aspects of the software system that is being developed [5]. Unlike external DSL, using the grammar of the selected language could lead to some obvious constraints. Thus, the less flexible the constructions of base language are, the less usable and effective the internal DSL is going to be. It means, when choosing the language, the capabilities of one have to correspond to the scope and use of the internal DSL which will be created on its basis. Last, but not least, with all of the functionality of the selected language as a base, the developer receives a ready-made set of development support tools, such as modern IDEs, plugins, documentation and so on. So, the developer loses complete freedom of definition of the grammar, remaining within the grammar of the base language, but at the same time gets the opportunity to use all the already available benefits of this language.

Another approach to creating internal DSL is the use of programming languages with configured syntax, i.e., languages focused on metaprogramming techniques. This approach is called "Extensible programming", which is a programming style focused on the use of mechanisms for expanding programming languages, translators, and execution environments [6]. The examples of these languages could be the follows: Forth, Common Lisp, Nemerle, and Racket.

As the syntax of all GPL is based on a text grammar, such grammars have one essential disadvantage: when it comes to expanding, it could become ambiguous [6]. In other words, there may be a situation when the same lines of source code will have several interpretations, and it’s really unknown, which one it is meant to be. This problem is especially visible when the developer is trying to combine several different grammar extensions into one language. Of course, separately they are absolutely unambiguous, but when combined together, it may lead to serious problems and further use of the language will be impossible. The refusal to use text grammar may be a possible solution. In this case, the program could be considered as an instance of the active syntactic metamodel. Usually, the metamodel of programs is presented in the form of an abstract syntactic tree. The examples of these ones could be follows: Scheme, Clojure, etc.

2.2 Software tools for DSL development

All language support tools, in fact, are the tools that not only help to create DSL, but also provide its elaboration as modern intelligent development environments, providing opportunities to build modern IDEs for the created languages. Such DSL development environments will be able to provide some essential capabilities, without which modern software development is impossible:

- Code auto-completion;
- Default automatic code generation;
- Tools for easy and flexible refactoring;
- Debugging of DSL scripts or scenarios;

- Integration with version control tools (git, cvs, svn);
- Unit and integration testing.

There are some frameworks for already existing popular editors, e.g., IntelliJ IDEA, however, it can't provide an appropriate level of support and integration with DSL. So, one of the most suitable solutions for this problem may be use of JetBrains MPS [7]. This is a metaprogramming system that implements a paradigm of language-oriented programming. It can be both an environment for language development and at the same time IDE for the developed languages.

In order to maintain the compatibility of language extensions with each other, MPS deal with the programs not as text, but as a syntax tree. This allows editing to take place directly. As a result, instead of specific language syntax, the MPS defines an abstract syntax (syntax tree structure) for the DSL, which is currently developed. MPS offers a special projection editor to work with the trees. It means for each node of the syntax tree, IDE creates the part of the screen, with which user can interact.

Summarizing the overview results given below it is possible to propose the classification scheme of the methods and tools for design and support of DSL compilers which are shown in Figure 2:

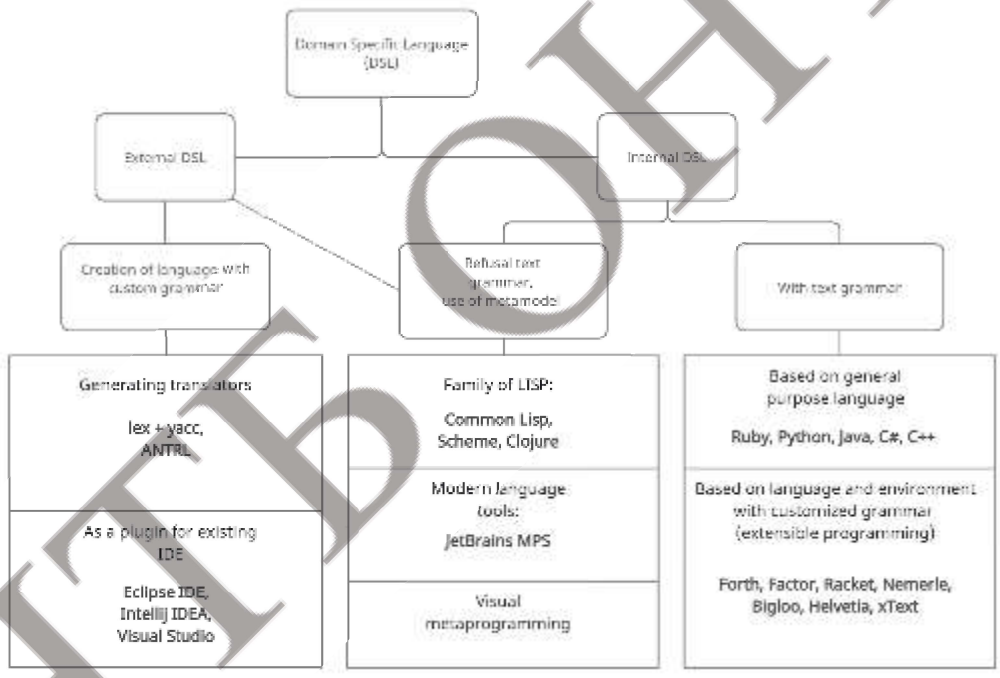

Fig. 2 – The proposed classification of methods and tools for creation and support of DSL

III. IMPLEMENTAION OF THE PROPOSED DSL COMPILER FOR ADAPTIVE SHA DEVELOPMENT

Taking into account this elaborated classification scheme we decided to elaborate the target DSL compiler for adaptive SHA development as an internal DSL with textual grammar using GPL Python and C++.

3.1 Grammar rules for the DSL

Any programming language (GPL or DSL) is a subset of the real (natural)

language and is created to facilitate and support the process of human communication with the computer. The compilation theory is built on the fact that any language can be described formally. Formally means that such a language consists of a set of finite words and their grammatical constructions. The syntax of a programming language, or an appropriate grammar is a collection of structure-corrected and pre-determined combinations of characters, which can be simply called as rules. The syntax of programming languages is usually defined with using of a combination of some regular expressions for its lexical structure and the Backus – Naur notation [8].

So, to define the syntax of DSL grammar rules for SHA, it is necessary to apply some special notation symbols, namely:

- {} – zero or more than zero,
- [] – zero or one,
- + – one or more than one from the left part,
- () – for grouping purpose.
- | – logical OR.

Some predefined words in the grammar rules can be whether links for other grammar rules or appropriate tokens [14].

Further, it is important to define the main grammar rule, without which there is no further grammar development possible. So, in this example, it will be dome as follows:

program :: {statement} (1)

The expression (1) means, that there is a grammar rule with the name “program”, which consists of zero or more than zero “statements”. In this case, a “statement” is another grammar rule, and it can be structured as the following set of the basic rules (see the expressions (2)-(6) respectively):

statement :: “DISPLAY” (expression | string | array | object) nl (2)

| “IF” comparison “THEN” nl {statement} “ENDIF” nl (3)

| "DECLARE" ident "=" expression nl (4)

| "INPUT" ident nl (5)

| ident "=" expression nl (6)

where “DISPLAY”, “IF”, “THEN”, “ENDIF”, “DECLARE”, “INPUT” are the appropriate keywords of the proposed DSL grammar rules with respect to typical process control algorithms used in SH applications; “string|”, “array”, “object”, “ident” are some variables of the different data types; “nl”, “expression”, “comparison” are other grammar rules, see below the definitions (7)-(12).

Other defined DSL grammar rules look like as follows:

nl ::= '\n'+ (7)

comparison ::= expression (("==" | "!=" | ">" | ">=" | "<" | "<=") expression)+ (8)

expression ::= term {("-" | "+") term} (9)

term ::= unary {("/" | "*") unary} (10)

unary ::= ["+" | "-"] primary (11)

primary ::= number | ident (12)

As it may be seen from the set of DSL grammar rules given in (1) - (12), a tree-like grammar structure will be generated using these ones sequentially. It provides an ability to construct the DSL expressions correctly, and the main implementation issues for the DSL compiler for these grammar rules are presented in the next subsection.

3.2 Software implementation of the main compiler' blocks

The first module of the compiler, which is the lexical analyzer (Lexer), will produce a stream of tokens. To do so, first what needs to be done, to implement the ability to track the current position in the input DSL text and character, which corresponds to this position. It will allow the compiler to analyze every symbol or set of symbols separately and find out which token it is. Of course, it is also needed to move the current position further and update the symbol on this position accordingly. In some cases (it will be explained later), it will be needed to know the next symbol without updating the current position. The code examples of these functions are shown in Figure 3:

```
def nextChar(self):
    self.curPos += 1
    if self.curPos >= len(self.source):
        self.curChar = '\0'
    else:
        self.curChar = self.source[self.curPos]

def peek(self):
    if self.curPos + 1 >= len(self.source):
        return '\0'
    return self.source[self.curPos + 1]

def abort(self, message):
    sys.exit("Lexing error. " + message)

def skipComment(self):
    if self.curChar == '#':
        while self.curChar != '\n':
            self.nextChar()

def skipWhitespace(self):
    while self.curChar == ' ' or self.curChar == '\t' or self.curChar == '\r':
        self.nextChar()
```

Fig. 3 – Python code fragment for lexical analysis

One of the main steps in creating a lexical analyzer is defining tokens, which will be allowed in the proposed compiler. A list of available tokens for adaptive SHA was received from the FODA model description (see in Figure 3). However, there are some tokens given by default:

- Operator – one or two characters: + - * / = != > >= < <=;
- String – quotation marks followed by zero or more characters;
- Number – one or more numeric characters followed by optional decimal part;
- Identifier – alphabetic character followed by zero or more alphanumeric characters;
- Keyword – some set of characters reserved by programming language.

There is a possibility of lexical errors in input DSL code, so it's needed to define a mechanism of handling such situations. So, in case if the lexical analyzer can't determine which token the current character is, it will prematurely complete the compilation process and notify the user about the lexical error. It is also important to skip all comments and non-used whitespaces, which may be present in the input text.

For example, for a mathematical operator recognition, it may be enough to analyze the current character and, if it is matched, the token is successfully identified. However, this approach can't recognize operators, which consist of 2 symbols, such as !=, >=, <=. So, for this type of operators, if the current operator could be 2-symboled, it's needed to check the following symbol (see in Figure 2).

The second module of the compiler is a Parser, which is directly connected with language grammar, so, the main goal is to implement language rules in the programming language. It means, that each rule in the formal grammar must have an appropriate handler in the parser. The input of the Parser is a stream of tokens, which were generated in the previous step. As well as in Lexer for symbols, for the Parser to work properly it is needed to track the current token and move to the next one after processing it. So, basically, the program will iterate on the token list and call the appropriate handler on each token match. The code examples of these handlers are shown in Figure 4:

```
def nl(self):
    self.match(TokenType.NEWLINE)
    while self.checkToken(TokenType.NEWLINE):
        self.nextToken()

def expression(self):
    self.term()
    while self.checkToken(TokenType.PLUS) or self.checkToken(TokenType.MINUS):
        self.emitter.emit(self.curToken.text)
        self.nextToken()
        self.term()

def term(self):
    self.unary()
    while self.checkToken(TokenType.ASTERISK) or self.checkToken(TokenType.SLASH):
        self.emitter.emit(self.curToken.text)
        self.nextToken()
        self.unary()

def unary(self):
    if self.checkToken(TokenType.PLUS) or self.checkToken(TokenType.MINUS):
        self.emitter.emit(self.curToken.text)
        self.nextToken()
    self.primary()
```

Fig. 4 – Python code fragment for parser`s handler of the rules (7-11)

It is important to understand that to achieve different levels of priority it is needed to consistently organize grammatical rules. In other words, operators with higher priority must be lower in grammar in order for them to be lower in the parsing tree, which is the output of the parser. Thus, operators, which are the closest to the tokens in the parsing tree (i.e. closest to tree leaves) will have the highest priority. Since binary operators "+" and "-" are lower in grammar rules, they will have higher priority than * and /. For a better understanding of this concept, let's consider simple math examples, which are 1 + 2*3 and 4 / -5. The generated parsing trees for these examples are shown in Figure 5:

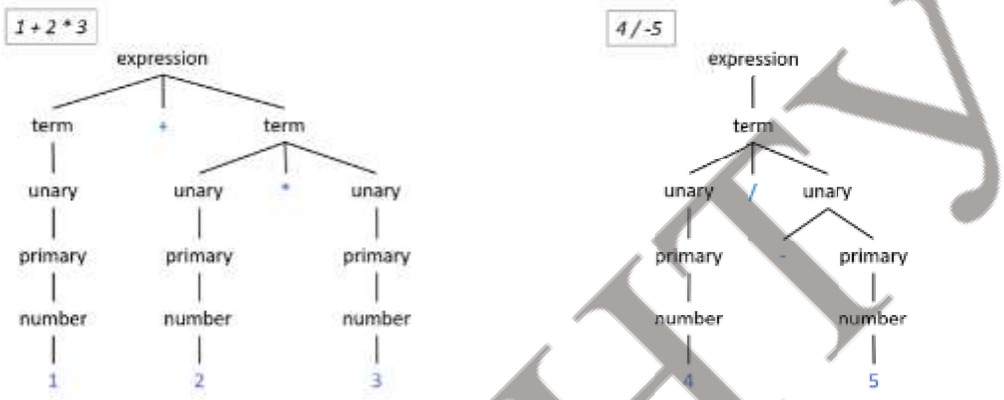

Fig. 5 – Generated parsing trees for some simple math expression examples

It means, that the multiplication operator will always be lower in the tree than the plus operator. The single negation operator (!) will be even lower. If there is more operators with the same priority, then they will be processed from left to right.

The last module of the compiler is a Code emitter. It will iterate along a parsing tree and for each handler function generate the corresponding C++ code. The generation of machine-executable code can be achieved by using any standard C ++ compiler. The usage of such an approach does not require to provide a code optimization by the created DSL compiler, as, in fact, this will be handled by the compiler of the source programming language. The code examples of generating target code are shown in Figure 6:

```
def program(self):
    self.emitter.headerLine("#include <stdio.h>")
    self.emitter.headerLine("#include <iostream>")
    self.emitter.headerLine("#include <string>")
    self.emitter.headerLine("using namespace std;")
    self.emitter.emitLine("int main(void){")

    while self.checkToken(TokenType.NEWLINE):
        self.nextToken()

    while not self.checkToken(TokenType.EOF):
        self.statement()

    self.emitter.emitLine("return 0;")
    self.emitter.emitLine("}")
```

Fig. 6 – Python code fragment of generating target C++ code

In Figure 6 proposed compiler emits base C++ file structure and some additional required libraries.

IV. RESULTS OF WORK IN A FORM OF PROPOSED QUANTITATIVE METRICS

To prove an effectiveness of the elaborated compiler for SHA development, it is needed to choose the specific quality metrics of software development. In this case, it was chosen the method of estimating the number of lines of code (LOC). Thus, one of the possible metrics of efficiency of the DSL compiler Kef (1) can be calculated by the formula:

$$Kef(1) = \frac{LOC_{DSL}}{LOC_{GPL}} * 100\%, \quad (13)$$

where LOC_{DSL} – the number of DSL lines of code; LOC_{GPL} – the number of generated GPL lines of code.

For the described specific use case, this value by formula (13) is calculated as 13/39*100% = 33%.

Another way to evaluate the quality of the created compiler is to compare the amount of C++ code generated by DSL compiler with the amount of C++ code that was created manually, for the same example of air conditioner controller. This example was found in the public code repository on the GitHub service [9].

Therefore, the second possible metric of the efficiency of the DSL compiler Kef (2) can be calculated by the formula:

$$Kef(2) = \frac{LOC(C++)_{GPL} - LOC(C++)_{DSL}}{LOC(C++)_{GPL}} * 100\%, \quad (14)$$

where $LOC(C++)_{DSL}$ – is the number of LOC generated by DSL compiler; $LOC(C++)_{GPL}$ – is the number of LOC in the C++ program written in a manual mode. For the described specific test case, this value calculated by the formula (14) is equal: (25-23)/25*100% = 8%.

It is to mention that the Kef (1) determines the advantage of the use of the DSL compiler from the point of view on cost reduction for the implementation of the resource management system of SHA. The Kef (2) determines the advantage of the use of the DSL compiler from the maintenance, support, and refactoring code point of view in the target SHA system.

In order to calculate the estimated weighted average efficiency score of the developed DSL compiler, it's needed to choose some so-called software development and maintenance importance factors. Let's consider them, e.g. as 0.35 for the Kef (1), and as 0.65 for the Kef (2) (in more correct way it can be done using one of the expert estimation methods, e.g, the Analytic Hierarchy Process [10]). Therefore, the final average value of the estimation metric K_avg can be calculated with the following formula:

$$K_{avg} = \left(0.35 * Kef(1) + 0.65 * Kef(2)\right) * 100\% = 16.75\% \qquad (15)$$

where Kef(1) and Kef(2) are the values of the compiler efficiency metrics calculated using the formulas (13) and (14) accordingly.

So, as a final result, the approximated weighted average efficiency score of the developed DSL compiler is equal to 16.75% that corresponds with some data about these issues already published (see, e.g. in [11]).

The additional ideas and more specific issues of this research can be found in [12].

V. CONCLUSIONS

In this paper we have motivated an actuality to apply a concept of domain-specific language (DSL) in such modern and complex problem domains as Internet of Things (IoT) systems and "Smart-Home" applications. The performed overview of the main methods and software tools for DSL design and implementation allowed us to elaborate their possible classification scheme and choice the appropriate way for our DSL development. The main functional blocks for the proposed DSL compiler are designed and implemented using Python and C++, and the effectiveness estimation for this compiler is done with calculation of two quantitative metrics that allowed to get the approximated weighted average about 16.75%. These results show the acceptable quality of the elaborated DSL compiler, and it allows to make the positive conclusions about the proposed approach.

Further work in this research is supposed to expand the grammar of the DSL compiler with special rules that will support effectively the variability of software components in "Smart-Home" systems, and to develop a comprehensive methodology for a performance evaluating of a prospective DSL compiler, taking into account the possible costs of its construction and usability for its end users.

VI. REFERENCES

1. D. Pandit, S. Pattanaik, "Software Engineering Oriented Approach to Iot - Applications: Need of the Day", in International Journal of Recent Technology and Engineering (IJRTE), Vol.7, Issue-6, 2019 - pp. 886-895.2.

2. M. Mazzara, I. Afanasyev, S. R. Sarangi, S. Distefano, V. Kumar and M. Ahmad, "A Reference Architecture for Smart and Software-Defined Buildings," 2019 IEEE International Conference on Smart Computing (SMARTCOMP), Washington, DC, USA, 2019, pp. 167-172, doi: 10.1109/smartcomp.2019.00048.

3. D. Karagiannis, H.C. Mayr, J. Mylopoulos, Domain-Specific Conceptual Modeling: Concepts, Methods and Tools. Springer, Berlin (2016).

4. R. Huber, L. Pueschel, M. Roeglinger, "Capturing smart service systems: Development of a domain-specific modelling language", in Inf. Systems Journal, Vol. 29, Issue 6 November 2019, pp. 1207-1255.

5. T. Parr, ANTLR, ANother Tool for Language Recognition. URL: http://www.antlr.org.

6. M. Voelter, S. Benz, C. Dietrich, MDSL Engineering: Designing, Implementing and Using Domain-Specific Languages, 2013.

7. JetBrains MPS, MetaProgramming System. URL: https://www.jetbrains.com/mps/

8. R. Wilhelm, H. Seidl, S. Hack, Compiler Design: Syntactic and Semantic Analysis. Springer-Verlag Berlin Heidelberg, 2014. doi: 10.1007/978-3-642-17540-4.

9. Xavier, Air_conditioning_control, 2015. URL: https://github.com/jeffersonxavier/air_conditioning_control

10. I.B. Botchway, A. E. Akinwonmi, S. Nunoo, Evaluating Software Quality Attributes Using Analytic Hierarchy Process (AHP), Journal of Advanced Computer Science and Applications, Vol. 12, No. 3 (2021) 165-173. doi: 10.14569/IJACSA.2021.0120321.

11. A. Barišić, V. Amaral, M. Goulão, Quality in Use of Domain Specific Languages: A Case Study // Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU '11), USA, Oregon, October 2011. – pp. 65–72.

12. Rustam Gamzayev, Mykola Tkachuk, Oleksandr Nelipa. Domain-Specific Language for Adaptive Development of "Smart-Home" Applications // Proceedings of the 1st International Workshop on Information Technologies: Theoretical and Applied Problems 2021 (ITTAP-2021) Ternopil, Ukraine, November 16-18, 2021. (http://ceur-ws.org/Vol-3039/) – pp. 154-165.