Domain-Specific Language for Adaptive Development of "Smart-Home" Applications

Rustam Gamzayev[a], Mykola Tkachuk[a] and Oleksandr Nelipa[a]

[a] *V.N. Karazin Kharkiv National University, Majdan Svobody 6, Kharkiv, 61077, Ukraine*

Abstract

The actuality to apply a domain-specific language (DSL) in such modern and sophisticated problem domains as the Internet of Things systems and "Smart-Home (SH)" applications is motivated. The briefly 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 new approach to DSL compiler designing for adaptive software development in SH applications is proposed which is based on a feature-oriented domain modeling for a configurable grammar rules construction. 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 estimation for this compiler is done with calculation of two quantitative metrics that allowed to get the approximated weighted average value about 16.75%. These results show the acceptable quality of the elaborated DSL compiler, allow to make some positive conclusions about the proposed approach, and formulate further steps to be done in this research.

Keywords 1

Domain-specific language, compiler, smart-home, adaptation, software, effectiveness, metric

1. Introduction

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]. The main final goal of all these approaches to IoT and SHA development is to reduce project's time and costs with respect to the appropriate quality requirements to be met in the target system solutions.

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 utilize of 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 and tools 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. The rest of our paper is organized as follows: Section 2 includes a short overview of some existing grammar models and software tools to create DSL for IoT and SH applications,

ITTAP'2021: 1nd International Workshop on Information Technologies: Theoretical and Applied Problems, November 16–18, 2021, Ternopil, Ukraine

EMAIL: rustam.gamzayev@karazin.ua (A. 1); mykola.tkachuk@karazin.com (A. 2); xa11867675@student.karazin.ua (A. 3)

ORCID: 0000-0002-2713-5664 (A. 1); 0000-0003-0852-1081 (A. 2); 0000-0003-1387-1414 (A. 3)

Section 3 provides an approach to DSL compiler designing for adaptive software development in SHA, especially, using feature-oriented modeling for the grammar rules construction. Section 4 presents the software implementation facets of the main DSL compiler's components, the first testing results, and introduces the quantitative metrics to effectiveness estimation of its usage. The paper concludes with Section 5 giving the brief summary and outlook on future work, and Section 6 provides the list of the information sources used in this research.

2. Briefly overview of some existing methods and tools to create domain-specific languages (DSL)

To elaborate an effective DSL compiler for adaptive SHA development it is necessary to analyze some existing methods and software tools for creation of DSL. The main results of this study are presented in this Section below.

2.1. Basic methods for DSL designing

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. Some 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 is shown in Figure 1:

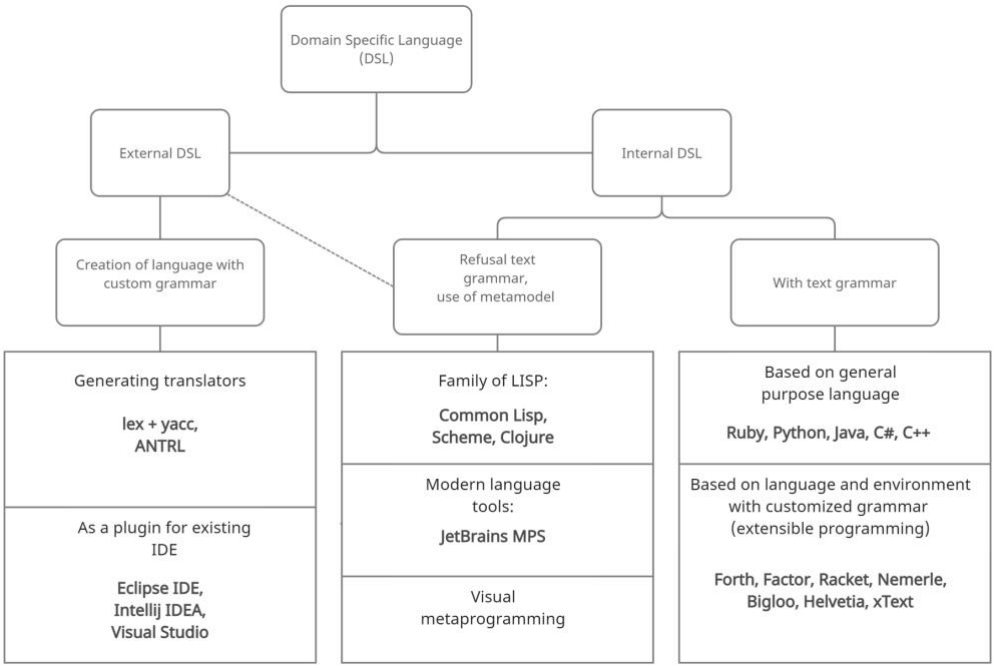

Figure 1: The proposed classification of methods and tools for creation and support of DSL

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. Design of DSL compiler for adaptive SHA development

In some recent publications dedicated to the problem of effective DSL's design and implementation for complex application domains, in particular, for IoT and SHA, the need to use appropriate model-driven approaches is emphasized (see e.g. in [8, 9,10]). That is why we propose to consider one of the possible domain modeling methods for SHA development that allows to provide target adaptive software solutions taking into account the resources variability issues in these projects.

3.1. Feature-oriented domain modeling for SHA

The first important problem of the real-life SHA development is a necessity to combine specific components of different vendors into one automated management system, and the second one is to provide an affective support for variability (or adaptability) of software components, with respect to permanent changes of different user requirements in such systems.

To identify specific SHA features, the initial domain model was created in FODA (Feature-oriented Design Analysis) notation [11], which is shown in Figure 2, and this one identifies all corresponding sensors and actuators aggregated into the different subsystems (provided, e.g., by Google, Amazon, Xiaome, etc.). All these devices are controlled by the special central control unit "HASystem", but for the correct support of whole SHA an appropriate separate hub for each of the subsystems is needed.

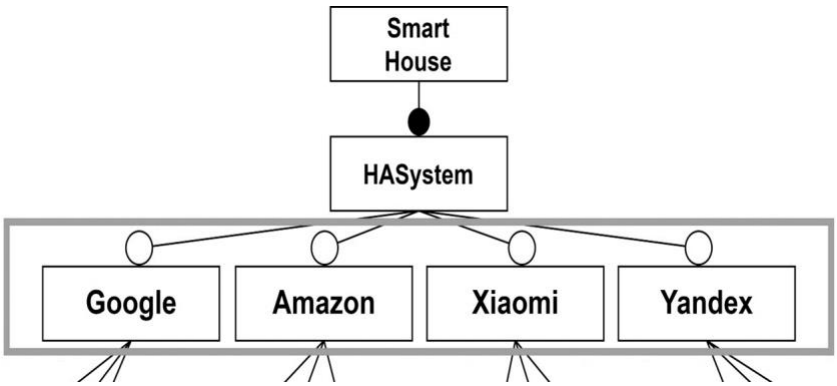

Figure 2: Initial FODA-model for SH applications

To develop the more sophisticated FODA-model taking into account the specific features of soft- and hardware components used in whole SHA it is necessary to handle the expert's knowledge and different user needs in this domain. One of the possible approaches to solve these problems is the usage of a repertory grids (RG) method [11], and the elaboration of special knowledge-oriented IT-technology for this purpose [12]. The target FODA – model is presented in Figure 3, and in terms of the RG-method it is given as the ontology-oriented graph which is built basing on the 2 sets:

a) a set of *Elements (E)*: e.g. *"Device", "TV", "Lamp", ...*); they can be defined based on the domain expert interviews which have to be divided into semantic objects to be represented as RG columns,

b) a set of *Constructs (C):* e.g. "*turned on / turned off*"; "*isUsedCondition*", "*should be switch on / switch off*", *...),* they characterize the alternative variable features of SH application.

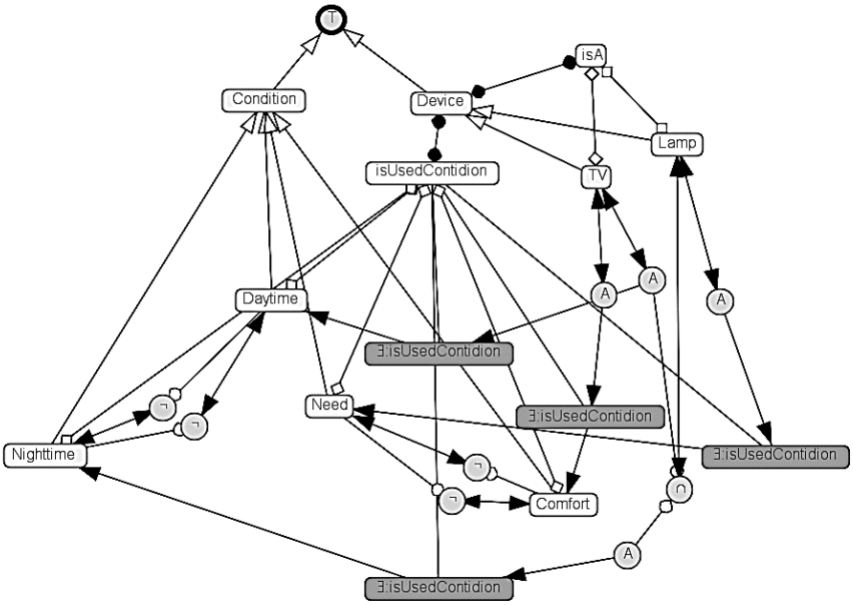

Figure 3: The target FODA-model for our SH application

Taking into account the semantical meaning of these 2 sets: *E* and *C* respectively, it is possible to utilize them for the configuring of the DSL to be designed for the adaptive SHA development.

3.2. Model-driven approach to designing of DSL for adaptive SHA development

Following the already mentioned trend to apply a model-driven approach to DSL design, we propose the formalized definition of DSL for the adaptive SHA development, denoted below as DSL(SHA), as a tuple

$$DSL(SHA) = \langle FODA(SHA),\ Grammar\ (Lexer,\ Parser),\ Code\_Generator \rangle, \quad (1)$$

where: *FODA(SHA)* is a feature-oriented model which reflects variability issues in the given application domain (see above in Section 3.1); *Grammar (Lexer, Parser)* is a set of models, algorithms, data resources, and software components which are used in order to implements all grammatical services (rules) for the given DSL, including a lexical analysis (*Lexer*) and a syntax parsing (*Parser*) for the input source code; *Code_Generator* is a code engine that converts a syntactically and semantically correct DSL code into a sequence of the programming instructions that can be executed by a target computer.

It is important to mention that the ontological model *FODA(SHA)* in formula (1), especially, its 2 sets given as (a)-(b), should be used for handling of tokens in Lexical analyzer in an adaptive way, because all necessary SHA functional features can be changed dynamically on this model's level. All other design and implementation issues related to the DSL(SHA) should be considered in other special research, but exactly in this paper we only elaborated the first vision for its technological scheme in IDEF0 notation [13] shown in Figure 4. It consists of 3 functional blocks (FB): "Lexical analysis A0", "Syntax analysis A1", and "Code generation A2". Each FB, corresponding to the IDEF0 notation rules, has 4 kinds of external interfaces: *Inputs* are the arrows enter into a given FB from the left side, they represent the data to be processed in this block; *Controls* are the arrows enter from the top, they define all business logic algorithms and models for this FB; *Mechanisms* are the arrows enter from below, and they reflect all implementation issues related to this FB (including human actors, if needed); *Outputs* are the arrows leave the FB from the right, and they are the results of data processing in this block.

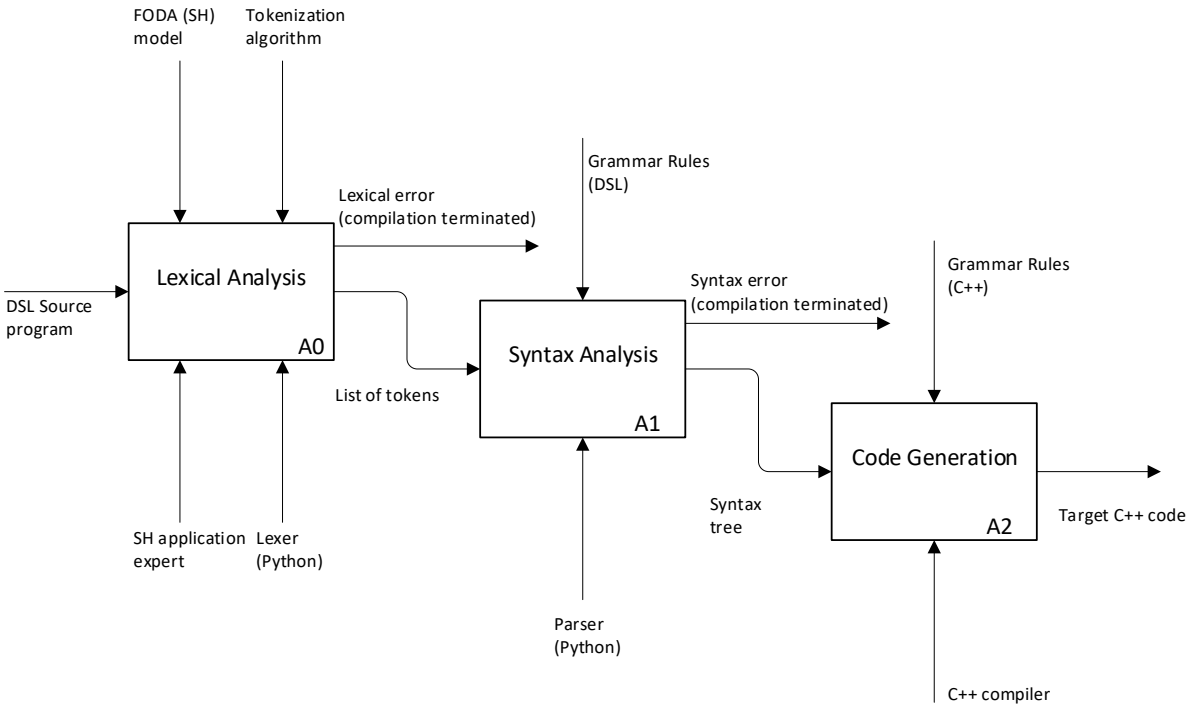

Figure 4: IDEF0 diagram for the proposed DSL compiler

The FB A0 performs the lexical analysis using as the constructed *FODA (SHA)* model (see in Figure 2) which includes all terms and constructs that can be used as the initial set of tokens for a target DSL. To process these tokens in correct way an appropriate tokenization algorithm has to be used, and to support this functionality the SHA expert operates with a *Lexer* module implemented with Python. As the correct output of this FB A0 the list of selected tokens is created that services as the input for the next FB A1, and in case of any lexical errors occurred the compilation process is terminated.

The FB A1 provides the syntax analysis using the grammar rules elaborated for the given DSL (see below in paragraph 3.3), and basing on the list of input tokens it creates the syntax-tree of the initial DSL program, if there are some syntax errors then the compilation process is terminated also. As the implementation mechanism for the FB A1 the *Parser* module written in Python is utilized.

Finally, the FB A2 gives us a target output as a C++ source code which is generated with respect to the grammar rules of this programming language, and any available C++ compiler can to be used as a correct tool for this purpose.

3.3. Grammar rules for the DSL with respect to SHA specific features

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 [14].

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} , (2)

The expression (2) 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 (3)-(7) respectively):

```
statement :: “DISPLAY” (expression | string | array | object) nl          (3)
| “IF” comparison “THEN” nl {statement} “ENDIF” nl                        (4)
| "DECLARE" ident "=" expression nl                                       (5)
| "INPUT" ident nl                                                        (6)
| ident "=" expression nl                                                 (7)
```

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 (8)-(13).

Other defined DSL grammar rules look like as follows:

```
nl ::= '\n'+                                                                (8)
comparison ::= expression (("==" | "!=" | ">" | ">=" | "<" | "<=") expression)+   (9)
expression ::= term {( "-" | "+" ) term}                                    (10)
term ::= unary {( "/" | "*" ) unary}                                        (11)
unary ::= ["+" | "-"] primary                                               (12)
primary ::= number | ident                                                  (13)
```

As it may be seen from the set of DSL grammar rules given in (2) - (13), 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 Section 4.

4. Software prototyping, testing, and an effectiveness assessment of the proposed DSL compiler

The proposed DSL compiler will be implemented with the usage of Python programming language [15]. In the text below, it will be demonstrated how to create a lexical analyzer, parser, and code emitter with basic functions for SHA. The output of the compiler is generated C++ [16] code as SHA contains various microcontrollers, because the C ++ programming language is one of the most suitable tools for programming and configuring them.

4.1. 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 5:

```
class Lexer:
    def __init__(self, input):
        self.source = input + '\n'
        self.curChar = ''
        self.curPos = -1
        self.nextChar()

    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]
```

Figure 5: 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 5).

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.

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 6:

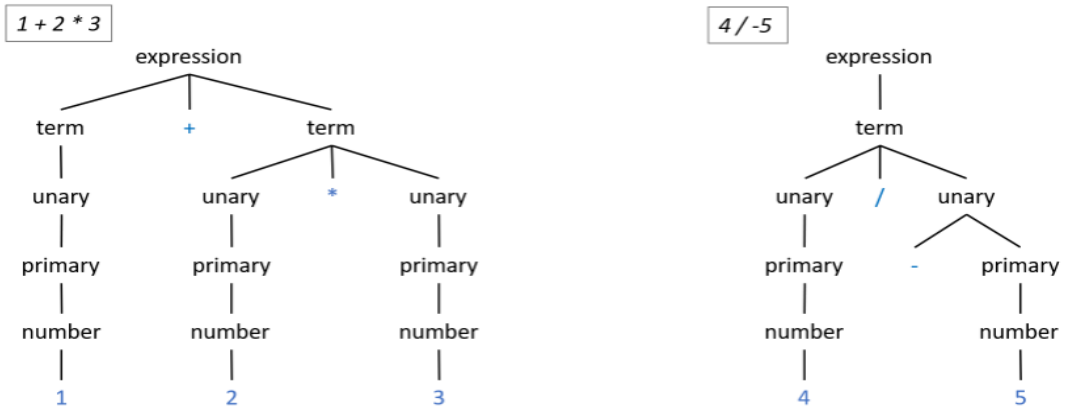

Figure 6: Generated parsing trees for some simple math expression examples

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.

4.2. The generated code examples for SHA

To analyze the performance of the created DSL compiler, let's consider a simple example of SHA: a room equipment with a smart air conditioner and temperature sensor. This sensor scans the room temperature, and if it is greater than a certain value, the air conditioner should be automatically turned on. This scenario with the help of DSL can be implemented as shown in Figure 7:

```
(1) INIT_TEMPERATURE_SENSOR sensor
(2) INIT_CONDITIONING conditioning
(3) INPUT sensor.homeTemp
(4)  IF sensor.homeTemp >= 30.5 THEN
(5)    conditioning.isTurnedOn = TRUE
(6)    conditioning.mode = 1
(7)    conditioning.currentTemp = 25.0
(8)  ENDIF
(9)  IF conditioning.isTurnedOn == TRUE THEN
(10)   DISPLAY "CONDITIONING TURNED ON"
(11) ELSE
(12)   DISPLAY "HOME TEMPERATURE IS NOT HIGH ENOUGH"
(13) ENDIF
```

Figure 7: The example of DSL program for testing SHA scenario

The operators (1), (2) initialize the sensor and the air conditioner variables. Further, the sensor outside (3) receives the room temperature value (just for example purposes, the temperature value will

be entered from the console, however in future extensions of DSL compiler, it is planned to implement the ability to received temperature by API call). On line (4) the temperature in the room is compared with a certain value. If the condition is valid, the air conditioner is turning on (5), its mode is being set to 1 (6), and the temperature it must maintain is being set at 25 degrees (7). C++ code generated by the compiler is shown in Figure 8:

```
#include <stdio.h>
#include <iostream>
#include <string>
using namespace std;
struct Temperature_Sensor {
    string name;
    string manufacturer;
    long id;
    bool isTurnedOn;
    double homeTemp;
    double outsideTemp;
};
Temperature_Sensor sensor;
struct Conditioning {
    string name;
    string manufacturer;
    long id;
    bool isTurnedOn;
    double currentTemp;
    double mode;
    bool timerIsTurnedOn;
    double timerTimeMS;
};
Conditioning conditioning;
int main(void) {
    cin >> sensor.homeTemp;
    if (sensor.homeTemp >= 30.5) {
        conditioning.isTurnedOn = true;
        conditioning.mode = 1;
        conditioning.currentTemp = 25.0;
    }
    if (conditioning.isTurnedOn == true) {
        cout << "CONDITIONING TURNED ON" << endl;
    }
    else {
        cout << "HOME TEMPERATURE IS NOT HIGH ENOUGH" << endl;
    }
    return 0;
}
```

Figure 8: Output C++ code generated by the DSL compiler

As can be seen in Figure 8, the generated C++ code fragment for described scenario contains 39 lines. On the other hand, the input DSL code contains 13 lines. Moreover, DSL has been designed in a way that the expert, which has some experience in configuring such systems, but no skills of use of high-level programming languages, will be able to use its functionality at a sufficient level to program SH applications using the developed compiler. It means, that the created software component will help to reduce costs and speed up some software development processes due to the lack of need for finding a programmer specializing in a particular programming language. Another advantage of using a DSL compiler is its functional flexibility and the ability to configure the various hardware and software configurations of the SH applications. In the case of requirements from experts, any lexical or syntax structures can be changed with a minimum spending time without affecting the end result.

4.3. Quantitative metrics and the first estimations for the elaborated DSL compiler effectiveness

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:

$$Kef(1) = \frac{LOC_{DSL}}{LOC_{GPL}} * 100\%\,, \tag{14}$$

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 (14) is calculated as $\frac{13}{39} * 100\% = 33\%$.

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 [17].

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

$$Kef(2) = \frac{LOC(C++)_{GPL} - LOC(C++)_{DSL}}{LOC(C++)_{GPL}} * 100\% , \quad (15)$$

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 (15) is equal: $\frac{25-23}{25} * 100\% = 8\%$.

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 of 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* [18]). Therefore, the final average value of the estimation metric K_{avg} can be calculated with the following formula:

$$K_{avg} = \big(0.35 * Kef(1) + 0.65 * Kef(2)\big) * 100\% = 16.75\% \quad (16)$$

where: *Kef (1)* and *Kef (2)* are the values of the compiler efficiency metrics calculated using the formulas (14) and (15) 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 [19]).

5. Conclusions and future work

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 new model-driven approach to DSL compiler designing for adaptive software development in SH applications is proposed, which is based on a feature-oriented domain analysis for a configurable grammar rules construction. 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.

6. List of 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] 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] B. Zarrin, H. Baumeister, An Integrated Framework to Specify Domain-Specific Modeling Languages. In Proceedings of 6th International Conference on Model-Driven Engineering and Software Development (pp. 83-94). SCITEPRESS Digital Library, (2018), doi:10.5220/0006555800830094.

[9] K. Babris, O. Nikiforova, U. Sukovskis, Brief Overview of Modelling Methods, Life-Cycle and Application Domains of Cyber-Physical Systems. In: Applied Computer Systems, Riga Technical University, May 2019, vol. 24, no. 1, pp. 1–8. doi:10.2478/acss-2019-0001.

[10] L. Buffoni, L. Ochel, A. Pop, P. Fritzson, Open Source Languages and Methods for Cyber-Physical System Development: Overview and Case Studies. In: Electronics 2021, 10, 902. doi:10.3390/electronics10080902.

[11] R.O. Gamzayev, M.V. Tkachuk, D.O. Shevkoplias, Handling of Expert Knowledge in Software Product Lines Development with Usage of Repertory Grids Method. In: Bulletin of V.N. Karazin Kharkiv National University, Series Mathematical Modeling, Information Technology. Automated Control Systems, 2020. - pp. 13-24.

[12] R.O. Gamzayev, M.V. Tkachuk, D.O. Shevkoplias, Knowledge-oriented Information Technology to Variability Management on Domain Analysis Stage in Software Development // // Advanced Information Systems. 2020. Vol. 4, No. 4, pp. 39-47. doi: 10.20998/2522-9052.2020.4.06

[13] I. Sommerville, Software Engineering, 10th Edition, Addison Wesley (2015).

[14] 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.

[15] Python Documentation. URL: https://www.python.org/doc/.

[16] Microsoft C++, C, and Assembler documentation. URL: https://docs.microsoft.com/en-us/cpp/?view=msvc-160.

[17] J. Xavier, Air_conditioning_control, 2015. URL: https://github.com/jeffersonxavier/air_conditioning_control.

[18] 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.

[19] 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.