The Object Model of the Subject Domain with the Use of Semantic Networks

Tetiana Kovaliuk[1’[0000-0002-1383-1589]’] and Nataliya Kobets[2’[0000-0003-4266-9741]’]

[1]National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute”, 37, Prospekt Peremohy, Kyiv 03056, Ukraine
`tetyana.kovalyuk@gmail.com`

[2]Borys Grinchenko Kyiv University, 18/2 Bulvarno-Kudriavska Str, Kyiv, 04053, Ukraine
`nmkobets@gmail.com`

Abstract. Object-oriented development of software systems and their components is based on the application of object-oriented models, technologies and tools that support them. Real systems are characterized by complexity, which is caused by problems describing the properties and behavior of subject domain objects. A description of the subject domain is given in the natural language, which can be considered an input language at the stage of object-oriented analysis. Syntax-oriented processing of sentences in the input language forms the output of the model software system. The conversion of the input language into a certain output is reduced to the construction of some translator, which, for any transformations of the sentences of the input language, uses the structure of this sentence. The ideas on which this concept is based include the ideas of formal grammars and semantic calculations for symbolic chains. These questions are the basis of the work under consideration.

Keywords: syntactic analysis, semantic analysis, semantic graph, classes, relations, object model

1 Introduction

Object-oriented programming (OOP) created a new style for programming systems by modeling subject domains (SD) with objects and their interfaces. The theoretical and applied conceptions of improved theory of design SD from objects of functional, interface data and isomorphism reflection of SD function of objects to the program components. [1]

We have a description of the subject environment in the natural language at the input. Our first step is to generate a report of the subject environment in the artificial language gen2, explicitly created to describe the subject environments. This is the notion stage. The solution is to statistically process the input data that do not have a clearly defined structure, and to generate the structured data that can be processed based on the classical algorithms of computational linguistics.

At the next stage, the generation of a semantic graph is performed based on the subject environment description in the language gen2. For this purpose a mathematical machine of context-free grammars is used.

The object model is generated once the semantic graph is constructed. At this stage, we apply the algorithm providing the use of design patterns and the results of the previously employed program runs. Generation based on the design patterns is completed; generation based on the previous program runs is in the stage of implementation.

The object model is stored in the form of structured information in the computer memory. There are no difficulties in generating software code or UML diagrams [2] [3]. At this stage, the frame code generation in C++ is implemented. The generation of results in other formats is a work in progress. After that a user can edit the model using the graphical interface. The database that contains the user's edits and that may be used for further runs is at the implementation stage.

2 Mathematical model of the problem

The language L is in development. This language allows to describe an object-oriented model. We can use the context-free grammar G to simplify the stage of the analysis of input data to determine the language L .

$$L = L(G) \tag{1}$$

A context-free grammar can be represented as a set of four objects [4] [5]:

$$G = (N, T, P, S) \tag{2}$$

where N is a finite set of nonterminal characters (or variavles) of the language, T is a finite set of terminal characters of the language, P is a limited set of production rules which are the rules for replacing nonterminal symbols. Production rules have the following form:

$$\alpha \to \beta, \alpha = (N \cup T)^{+}, \beta = (N \cup T) \tag{3}$$

where α are variables which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols, β are string of variables and terminals. S is the original grammar symbol from the set of non-terminal characters N

Now let us consider the stage of semantic analysis. Assign a set of valid identifiers I that include chains of Latin alphabet characters and digits whose length does not exceed 255 characters. The first character in a string must be a letter.

At the first stage of semantic analysis, we should construct a semantic graph S :

$$S = (V, E, \varphi) \tag{4}$$

where V is the set of vertices of the graph, E is the set of edges of a graph, φ is an incidence function

Let us define the functions for each vertex of the graph:

$$VertexName : V \to I \tag{5}$$

$$VertexType : V \to \{EssenceType, OperationType\} \tag{6}$$

Formula **Error! Reference source not found.** matches each vertex of a graph with a unique identifier. The expression **Error! Reference source not found.** specifies the type of vertex. Each vertex can describe the subject environment or the operation or process that occurs in the subject environment. According to the type of vertex, we can select two subclasses of the set of vertices:

$$\begin{aligned} &Essences = \{v \in V \mid VertexType(v) = EssenceType\} \\ &Operations = \{v \in V \mid vertexType(v) = OperationType\} \\ &Essences \cup Operations = V \\ &Essences \cap Operations = \varnothing \end{aligned} \tag{7}$$

Define the EdgeType function for the edges of the graph:

$$EdgeType : E \to EdgeTypes \tag{8}$$

$$EdgeTypes = \begin{cases} ContainEsType, CanType, UsesType, \\ \mathrm{Re}\,alIsType, IsType, DelegatesType, \\ DelegatesToType \end{cases} \tag{9}$$

The expression (8) specifies the graph type for each edge. The set of edge types (9) includes a significant number of elements. Let us split the set of edges into subclasses by type:

$$\begin{aligned} &\mathrm{Re}\,lations_i = \{e \in E \mid EdgeType(e) = EdgeTypes_i\}, i = \overline{1, \mid EdgeTypes \mid}, \\ &\bigcup_i \mathrm{Re}\,lations_i = E, \bigcap_i \mathrm{Re}\,lations_i = \varnothing \end{aligned} \tag{10}$$

The type of edge imposes certain restrictions on the type of vertices that are incident with it:

$$RoleA = EdgeTypes \to VertexType \tag{11}$$

$$RoleB = EdgeTypes \to Vertextype \tag{12}$$

$$\begin{aligned} &IsValid(e) = EdgeType(e) \equiv type \wedge \langle v_1 \mid v_2 \rangle \equiv \varphi(e) \wedge \\ &RoleA(type) \equiv VertexType(v_1 \wedge RoleB(type) \equiv VertexType(v_2) \end{aligned} \tag{13}$$

For each type of the relation, functions (11) and (12) determine the types of vertices to which the relation can be applied. The Boolean function (13) accepts the relation at the input and returns the logical truth at the output if the relation is validated. If (13) returns error, then the relation is set incorrectly and should be excluded from the graph.

Breakdown proceeds according to the classic syntax-directed translation scheme [6]. Syntactic parser initiates a lexical analyzer to receive the next non-terminal, and initiates semantic non-terminal procedures thereby guiding the semantics analysis.

In our case, the semantic procedures complement the semantic graph (4) with new vertices and edges. Each edge is validated against (13) before being added to the graph.

The object-oriented model begins construction after the semantic graph is made. This model is a set of classes, which can be described as (14):

$$Model = \{Class\}, Model \in M \tag{14}$$

Let $\mathbb{M}$ be the set of all object-oriented models. Each class can be represented as an ordered triple:

$$Class = \langle Name \mid \{Attribute\} \mid \{Method\}\rangle, ClassName \in I \tag{15}$$

where $Name$ is the name of the class from the set of valid identifiers I; $\{Attribute\}$ is a set of class attributes; $\{Method\}$ is a set of methods.

Define the class attribute as a pair:

$$Attribute = \langle Name \mid Type\rangle, Name \in I, Type \in Model \tag{16}$$

where $Name$ is the name of the attribute from the set of valid identifiers I, $Type$ is an attribute data type. It is one of the classes that are described in the $Model$.

The class method can be represented as a triplet:

$$\begin{gathered} Method = \langle Name \mid ReturnType \mid \{\langle ArgName \mid ArgType\rangle\}\rangle \\ Type, ArgumentType \in Model, Name, ArgumentName \in I \end{gathered} \tag{17}$$

where $Name$ is the name of the method from the set of valid identifiers I, $ReturnType$ is the data type of the value returned by the method; $\{\langle ArgName \mid ArgType\rangle\}$ is a set of method arguments, each represented by a pair, which includes the name of the attribute $ArgName$ and its data type $ArgType$.

Let us consider the algorithm, which is the basis for the semantic graph to construct the object-oriented model.

1. Build a set $Model$ by expression (14):

$$Model = \{VertexName(v), \varnothing, \varnothing \mid v \in Essences\} \tag{18}$$

2. For each vertex $v \in V$

a. Obtain all the edges E', adjacent to the vertex v.
b. Construct a partially ordered set $\langle E', \leq \rangle$, where relation of the partial order $"\leq"$ will allow determining the sequence where it is necessary to process the edges passing from this vertex.
c. $Model = ApplyRelation(e, Model)$, where $ApplyRelation : E \otimes M \to M$ accepts the arc of the semantic graph $e \in E$ with its model $m \in M$ at the input and returns model $m' \in M$, supplemented by additional classes, methods, attributes, and the relationship between classes at the output.

3 Stages of syntactic analysis

Syntactic analysis is performed using the top-down parsing algorithm. This section presents the employed grammar. The initial nonterminal character is represented by $start$. It displays nonterminal characters $sentence$ separated by a dot. After the last nonterminal of the sentence, the point can be applied selectively. The nonterminal symbol $sentence$ defines the sentence. A sentence in this context is a unit that describes one or another connection between the entities of the subject environment. There are such sentences as:

- $operations \to word\ space\ "can"\ space\ (singleOperation\ \%\ ',')$ which describes the actions that are inherent to the essence of the subject environment;
- $abstractOperations \to word\ space\ "specify"\ space\ (singleOperation\ \%\ ',')$ which describes the connection between entities, when one entity is a subclass of another;
- $generalization \to word\ space\ "is"\ space\ word$ is a description of the connection between entities, when one entity is a subclass of another;
- $implementation \to word\ space\ "implements"\ space\ word$ is a description of the connection between entities, when one entity can also be classified as another entity;
- $contains \to word\ space\ ("contains"\ |\ "has")\ space\ words$ is a connection when one or more entities are part of another entity;
- $composition$ is a complex connection that allows you to establish complex hierarchical relationships between the entities of the subject environment;
  $$composition \to "Composition"\ space\ word\ space\ "has"\ space\ words\ space\ "leafs"$$
  $$space\ "and"\ space\ words\ space\ "composites"$$
- $empty \to \varepsilon$ is an empty sentence that does not carry information content and is ignored.

Nonterminals are used to describe primary syntax components [6] [7]:

- $alpha \to [A-Za-z]$ is a letter;
- $alnum \to alpha\ |\ [0-9]$ is a letter or a digit;

- $space \to [\backslash n \backslash t]+$ –one or more indentation characters: space characters, tab characters, new line characters;
- $word \to alpha\ alnum*$ – word, a sequence of letters and digits, which should begin with a letter;
- $words \to word\ \%','$ – sequence of words separated by a comma.

The form in which this grammar is defined differs from the classical Backus-Naur notation [8]:

- the abbreviation $A\%B$ introduced for expressions of the form $A(B\ A)*$;
- regular expressions are specified instead of nonterminal characters;
- exit rules are arranged in descending order of priority.

4 Stage of semantic analysis. Construction of the semantic graph

The stage of semantic analysis is challenging to formalize [10]. This section deals with the language constructs of the gen2 language and describes the semantic actions that process them.

4.1 The generalization relation

The generalization relation syntax looks like:

$$A_1, A_2, \ldots A_n \text{ is } B \tag{19}$$

Generalization establishes a relation where one or several entities are subclasses of another entity. Inheritance of classes is an analog of generalization in the object-oriented programming [11] [12] [13]. The essence of this operation is to indicate that a specific entity can perform the same actions and have the same properties as another entity but at the same time possess its specific features.

The semantic graph is supplemented with vertices during processing of this design:

$$\begin{aligned} &v_1, v_2 : VertexName(v_1) = A, VertexName(v_2) = B, \\ &VertexType(v_1) = VertexType(v_2) = Essence \end{aligned} \tag{20}$$

and arc e:

$$\varphi(e) = (v_1, v_2),\ EdgeType(e) = IsType \tag{21}$$

In the second stage of the semantic analysis, we should add two classes A and B to the object-oriented model. It will be determined that A inherits from B .

Consider an example:

$$CoffeeTable\ is\ table \tag{22}$$

This relation indicates that the essence of the $CoffeeTable$ is a subclass of $table$. In fact, the coffee table has the features and characteristics inherent in the table, but it has its distinctive features, such as the purpose of keeping books, personal belongings and diminished dimensions. Figure 1 depicts a semantic graph corresponding to this relation.

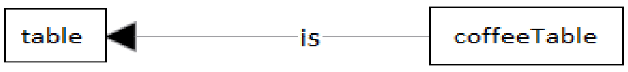

Fig. 1. Semantic graph for the generalization relation

4.2 The functionality relation

The syntax for determining the functionality is:

$$A\ can\ B_1(C_{11}, C_{12}, ..C_{1m}), B_2(C_{21}, C_{22}, ..C_{2m}), ..B_n(C_{n1}, C_{n2}, ..C_{nm}) \tag{23}$$

This relation determines that the subject environment includes entities A, $\{C_{ij}\}_{1,1}^{n,m}$, and the essence of A can perform actions of $\{B_i\}_1^n$ using the essence $\{C_{ij}\}_{1,1}^{n,m}$.

The vertices v_1, v_2, v_3, v_4 will be added to the semantic graph in the process of constructing this relation:

$$\begin{aligned} &VertexName(v_1) = A, VertexName(v_2) = B, VertexName(v_3) = C,\\ &VertexName(v_4) = op,\\ &VertexType(v_1) = VertexType(v_2) = VertexType(v_3) = Essence,\\ &VertexType(v_4) = Operation \end{aligned} \tag{24}$$

as well as arcs e_1, e_2, e_3 according to (25):

$$\begin{aligned} &\varphi(e_1) = (v_1, v_4), EdgeType(e_1) = CanType, \varphi(e_2) = (v_4, v_2),\\ &\varphi(e_3) = (v_4, v_3), EdgeType(e_2) = EdgeType(e_3) = UsesType \end{aligned} \tag{25}$$

At the second stage of the semantic analysis, three classes A, B, C will be added to the object-oriented model. Method op will be added to class A. This method accepts two arguments of types B and C.

Consider an example:

$$CoffeeMaker\ can\ makeCoffee(water,\ coffeeBeans) \tag{26}$$

This relation indicates that the entity $CoffeeMaker$ can prepare coffee while using water and coffee beans. Figure 2 shows a semantic graph corresponding to this relation

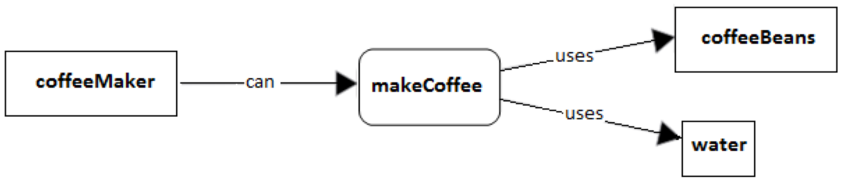

Fig. 2. Semantic graph of the functioning relation

4.3 Definition relation

The syntax of the definition relation has the form:

$$A_1, A_2, \ldots, A_n \textit{ implements } B \tag{27}$$

This relation indicates that a specific entity is capable of performing the functional duties of another entity. The class inheriting from the interface is an analogue of generalization in object-oriented programming [11] [12] [13]..

During processing, the semantic graph is supplemented with vertices v_1, v_2 :

$$\begin{gathered} VertexName(v_1) = A, VertexName(v_2) = B, \\ VertexType(v_1) = VertexType(v_2) = Essence \end{gathered} \tag{28}$$

and arc e :

$$e : \varphi(e) = (v1, v2), EdgeType(e) = ImplementsType \tag{29}$$

At the second stage of semantic analysis, class A and interface B will be added to the object-oriented model. It will be determined that A is inherited from B .

Consider an example:

$$MicrosoftWord\ implements proprietary Application, textProcessor \tag{30}$$

This relation (30) indicates that the $MicrosoftWord$ essence (the Microsoft Word product) can be regarded as $proprietaryApplication$ (proprietary software product) or $textProcessor$ (word processor). In fact, $MicrosoftWord$ has all the peculiarities inherent to both the proprietary software and a word processor. Figure 3 represents a semantic graph corresponding to this relation (27):

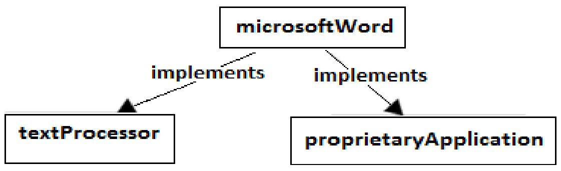

Fig. 3. Semantic graph for the generalization relation

4.4 The specification relation

The syntax for determining the ratio of the specification has the form:

$$A\ specify\ B_1\ (C_{11}, C_{12}, ...C_{1m}), B_2\ (C_{21}, C_{22}, ...C_{2m}), .., B_n\ (C_{n1}, C_{n2}, ...C_{nm}) \quad (31)$$

This relation (31) determines that the subject environment includes entities A and $\{C_{ij}\}_{1,1}^{n,m}$ while the entity A can perform actions of $\{B_i\}_1^n$ using the entity $\{C_{ij}\}_{1,1}^{n,m}$. However, there is no information on how these actions are carried out. Implementation of the specification relation differs from the implementation of the functionality relation by the fact that the semantic graph is supplemented with vertices v_1, v_2, v_3, v_4 during processing of its construction:

$$\begin{aligned} &VertexName(v_1) = A, VertexName(v_2) = B, VertexName(v_3) = C, \\ &VertexName(v_4) = op, \\ &VertexType(v_1) = VertexType(v_2) = VertexType(v_3) = Essence, \\ &VertexType(v_4) = Operation \end{aligned} \quad (32)$$

and with arcs e_1, e_2, e_3:

$$\begin{aligned} &\varphi(e_1) = (v_1, v_4), EdgeType(e_1) = SpecifyType, \varphi(e_2) = (v_4, v_2), \\ &\varphi(e_3) = (v_4, v_3), EdgeType(e_2) = EdgeType(e_3) = UsesType \end{aligned} \quad (33)$$

At the second stage of the semantic analysis, three classes A, B, C will be added to the object-oriented model. An abstract method op will be added to the class A, which accepts two arguments of B and C types.

Consider an example:

$$DataProvider\ \ specify\ \ getData\,(index) \quad (34)$$

Relation (34) indicates that the $DataProvider$ (data source) can provide information $getData$, while using the information identifier $index$. However, nothing is known about the nature of the information provided and about the mechanisms that will be used to collect this information.

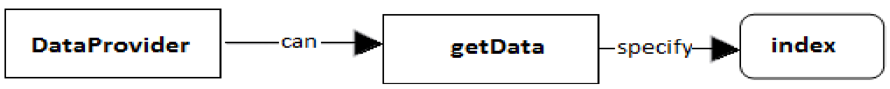

Fig. 4. Semantic graph for the ratio of the specification

4.5 Inclusion relation

The syntax for determining the inclusion is:

$$A\,contains\ B_1, B_2, \ldots, B_n \tag{35}$$

This relation (35) determines that the subject environment includes entities A and $\{B_i\}_1^n$, and entity A includes entities $\{B_i\}_1^n$ as its components.

During processing, the semantic graph is supplemented with vertices v_1, v_2 :

$$\begin{aligned} &VetrexName(v_1) = A, VertexName(v_2) = B, \\ &VertexType(v_1) = VertexType(v_2) = Essence \end{aligned} \tag{36}$$

and with arc e :

$$\varphi(e) = (v_1, v_2),\ EdgeType(e) = ContainsType \tag{37}$$

At the second stage of semantic analysis, two classes A and B will be added to the object-oriented model. Class A will contain a type B attribute.

Consider an example:

$$Guitar\ contains\ body,\ strings \tag{38}$$

This attitude indicates that the Guitar entity includes the body and strings entities. The figure 5 represents a semantic graph corresponding to this relation (38).

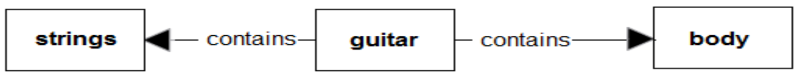

Fig. 5. Semantic graph for the inclusion relation

4.6 The composition relation

The composition relation is based on the design pattern “composition”. Design pattern “composition” describes a group of objects, considered by using the same interface. The task of this design pattern is to build a tree-like data structure that defines the hierarchy. The composition allows access to individual objects and their compositions using a unified interface.

Programmers often split vertex-leaves and vertex-nodes when working with tree-like data structures. It makes the code more complex and increases the probability of a mistake. The solution is to use an interface that allows access to complex and primitive objects in the same way. In the object-oriented programming, a composite object is an object consisting of one or more similar objects, each of which has similar functionality. This relation is known as "has-a". The key concept is that it is possible to manage a group of objects by managing only one object. The composition can be used when the client code should ignore the difference between complex and simple objects. It is because the code processes the former and the latter in the same way [11] [12] [13].

The composition relation syntax is:

$$\textit{Composition A has } B_1, B_2, .., B_n \textit{ leafs and } C_1, C_2, .., C_m \textit{ composites} \tag{39}$$

This relation (39) determines that the subject environment includes entities A, $\{B_i\}_1^n$ and $\{C_i\}_1^m$. According to the description above: A performs the role of Component, $\{B_i\}_1^n$ performs the role of Leaf, $\{C_i\}_1^m$ – the role of Composite.

During processing of this design, the semantic graph will be supplemented with vertices with names $A, B, C, AComposite$ describing entities and vertices with names $addChild, removeChild, child$, which describe operations.

Arches defining relations will be added:

- the entity $AComposite$ performs operations $addChild, removeChild, child$;
- the entity $AComposite$ defines A;
- the entity B defines A;
- the entity C defines $AComposite$;
- the entities A are involved in the $addChild, removeChild$ operation.

Consider an example:

$$\textit{Composition fileSystemObject has file,link leafs and directory composites} \tag{40}$$

File system element $fileSystemObject$ acts as a component of the composition. The simple elements of the composition are $file, link$. The complex element of the composition is $directory$. The figure 6 depicts a semantic graph corresponding to this relation (40):

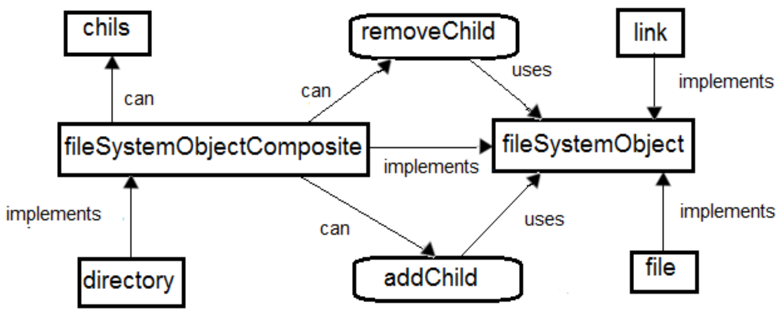

Fig. 6. Semantic graph for composition relation

4.7 Validation of a semantic graph

After constructing the semantic graph, it is necessary to validate it. If the model is in the form of a semantic graph, it is easy to notice logical errors. If we consider a semantic graph as a structure in the computer memory, we can use an algorithm on the graphs to search for logical errors committed during its construction.

First, check the Incompatibility of the types of arcs and types of edges. Sum up and give a list of types of edges and vertices. The following types of edges are used:

- ContainsRelation;
- ImplementationRelation;
- IsRelation;
- CanRelation;
- SpecifyRelation;
- DecoratesRelation;
- UsesRelation.

Two types of vertices are used:

- EssenseNode;
- OperationNode.

There are restrictions that arcs of a specific type can connect only vertices of a particular type. There are 4 classes of arcs according to this criterion:

- arcs that connect vertices of the EssenseNode type with the vertices EssenseNode: ContainsRelation, IsRelation, ImplementationRelation;
- arcs that connect vertices of the EssenseNode type with the vertices OperationNode: CanRelation, SpecifyRelation;
- arcs that connect the vertices of the OperationNode type with the vertices EssenseNode: UsesRelation;
- arcs that connect the tops of the OperationNode type with the vertices OperationNode: DecoratesRelation.

Therefore, validation provides checking of correspondence of arcs and vertices types.

In general case, a semantic graph may contain cycles. For example, an entity may contain an operation involving one or more entities of the same type. However, some types of connections should not be locked up. For example, there can be no loops formed by *IsRelation* -type arcs.

The following criteria are used for assessing the quality of the constructed semantic graph:

- number of unconnected areas in the graph: if the graph consists of two or more domains, then this indicates that the semantic model is in fact two models;
- the average number of edges entering a vertex: the higher the value of this indicator - the higher is the connectivity of the model;
- maximum number of edges entering or leaving a vertex;

- the ratio of the average number of edges entering the vertex to the maximum number of edges entering the vertex: if this value is significantly higher than 2, it indicates that the graph is not balanced;
- the ratio of the number of vertices to the number of edges.

5 Construction of an object model

After the semantic graph is constructed and the validation is executed, the object model is generated.The object model represents a hierarchy of objects describing the hierarchy of data types and their attributes and operations. This model is used to generate software code in the future.

This process represents the sequence of stages. The first steps are the graph walk and supplementation of the object model with information. In fact the data is copied and converted from one format to another. The final steps consist of the complex analysis of the semantic graph and the already constructed part of the object model. Based on the results of this analysis, the object model is supplemented with new information.

The construction of an object model includes the following steps:

- generation of data types based on the entities described in the semantic graph;
- addition of attributes to data types;
- addition of operations to data types;
- addition of information on basis and derivative classes to the object model;
- inheritance from the interfaces execution.

After that, if the object model is built, there is code generation. Code generation boils down to a recursive walk of the object model and creation of corresponding language constructs.

5.1 The comprehensive example

Let's have on input the following object model:

Composition furniture has sofa, chair, cupboard leafs and living Room, kitchen composites.
Furniture specify color, size and location.
Furniture has name. Location has x, y.

After the analysis is complete, a semantic graph is constructed. The result is shown in the figure 7.

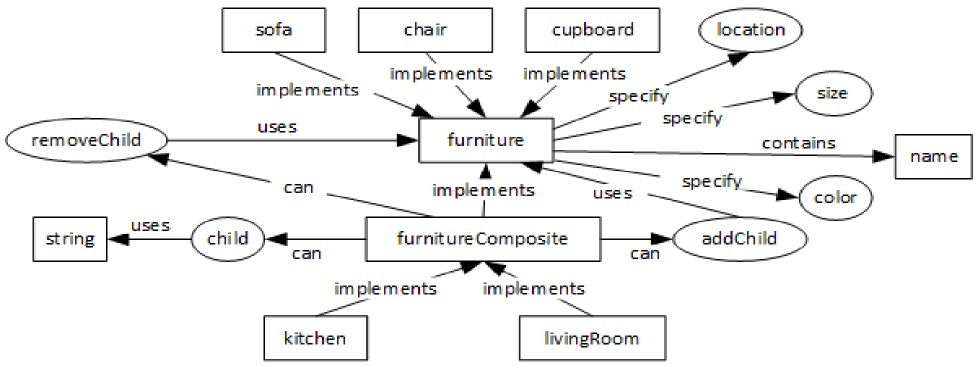

Fig. 7. Semantic graph on the control example

Generation of the object model is the next stage. At the final stage, the program code is generated.

6 Conclusion

The algorithm of automated creation of object-oriented models is considered. At the first stage, the description of the subject environment in the natural language is transformed into a description in the artificial language gen2. Then the text in the artificial language undergoes syntactic and semantic parsing. The semantic graph is based on the results of the parsing. This graph is validated for identifying logical errors. The object-model in the form of classes is based on the data obtained from the analysis of the semantic graph. An object model can further be used to generate software code, UML-diagrams, design documentation, etc.

References

1. Lavrischeva, E.: Object Modeling of Subject Domains, https://cyberleninka.ru/article/n/object-modeling-of-subject-domains, last access 2019/02/27
2. Fowler, M.: Patterns of Enterprise Application Architecture, Addison-Wesley, Pearson Education (2003)
3. Larman, C.: Applying UML and Patterns. In: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Prentice Hall (2004)
4. Hopcroft, J., Motwani, R., Ullman J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley publisher company (2002)
5. Rayward-Smith, V. J.: A First Course in Formal Language Theory. Blackwell Scientific Publications (1986)
6. Hulakov, V.V., Kovaliuk, T.V.: Modelyuvannya opysu predmetnoho seredovyshcha zasobamy syntaksychno-oriyentovanoyi translyatsiyi. In: Adaptyvni systemy avtomatychnoho upravlinnya, 12 (32), pp. 54-61 (2008)

7. Hulakov, V.V., Kovaliuk, T.V.: Stvorennya interpretatora stsenariyiv z metoyu rozshyrennya funktsionuvannya skladnykh system. In: Intern. Conference SNKPMI-2009, Lviv, 2009 (2009)
8. Backus–Naur form, https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form, last access 2019/02/20
9. Brooks, F.: No Silver Bullet – Essence and Accidents of Software Engineering. In: Computer Magazine, University of North Carolina at Chapel Hill (1987)
10. Sidorov, Yu.V., Leont'yev, A.A., Rogov, A.A., Zakharov, V.N.: Komp'yuternaya avtomatizirovannaya sistema dlya lingvisticheskogo razbora literaturnykh tekstov. In: IV Sankt-Peterburgskaya Assambleya molodykh uchenykh i spetsialistov, pp 66-72 (1999).
11. Suad Alagic: Object-Oriented Technology, Springer International Publishing Switzerland. (2015)
12. Satzinger, J.W., Orvik T.U.: The Object-Oriented Approach: Concepts, Systems Development, and Modeling with UML, Second Edition 2nd Edition. Cengage Learning; 2 edition (2001)
13. Young, B.J., Houston, K.A., Conallen, J., Engle, M.W., Maksimchuk, R.A., Booch, G.: Object-Oriented Analysis and Design with Applications, Third Edition. Addison-Wesley Professional Publisher (2007)