Scientific Research Publishing

Journal of Software Engineering and Applications, 2025, 18(7), 217-248
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

UML Class Diagram Classification Using Category Theory

Alexey Tazin, Mieczyslaw M. Kokar

Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA
Email: tazin.a@northeastern.edu, m.kokar@northeastern.edu

How to cite this paper: Tazin, A. and Kokar, M.M. (2025) UML Class Diagram Classification Using Category Theory. *Journal of Software Engineering and Applications*, **18**, 217-248.
https://doi.org/10.4236/jsea.2025.187014

Received: May 21, 2025
Accepted: July 15, 2025
Published: July 18, 2025

Abstract

To effectively evaluate a system that performs operations on UML class diagrams, it is essential to cover a large variety of different types of diagrams. The coverage of the diagram space can be attained by grouping them into a finite number of disjoint equivalence classes, each containing diagrams that are structurally equivalent, and analyzing at least one diagram from each class. Currently, no formal method exists to implement this approach. In this paper, we describe an approach that implements this idea by mapping class diagrams into an appropriate UML category using category theory and defining isomorphism classes containing diagrams that are isomorphic. The main idea of our approach to partitioning the space of class diagrams is to consider that any UML class diagram is an instance of the UML metamodel, then identify a small number of basic template diagrams, carefully selected simple class diagrams with parameters/variables, and combine them into complex template diagrams using an operation based on the colimit from category theory. We demonstrate through experiments that almost all of a large set of class diagrams randomly generated from the UML metamodel can be classified as instances of the isomorphism classes generated through this combination operation.

Keywords

UML, Category Theory, Class Diagram Classification, Class Diagram Composition, Graph Isomorphism

1. Introduction

Various software engineering tools in the Model-Based Software Engineering manipulate UML Class Diagrams. These tools take class diagrams as input and produce either different kinds of class diagrams or develop other kinds of output. The

first kind is referred to as *refactoring*, while the other is called *transformation*. The reasons for refactoring can be to: correct errors, incorporate additional requirements, improve design clarity, improve maintainability, apply design patterns, remove redundancies, modify inheritance structures, simplify aggregations, introduce abstractions, reduce complexity, merge diagrams, partition, factor, and slice diagrams. Transformation, on the other hand, is used for code generation, database schema generation, and visual representation. Recently, machine learning has been used to perform various operations on class diagrams, such as model-driven prediction, multidiagram reasoning, or code generation from class diagrams. As is usually the case, machine learning requires diverse and representative training data.

To effectively evaluate a system that operates on UML class diagrams, it is crucial to test it against a broad range of structurally diverse diagrams. In the absence of a large and varied industrial dataset, software engineers must generate diagrams that capture common real-world design patterns. To avoid redundant and excessive testing, these generated diagrams can be grouped into structural equivalence classes. Instead of testing thousands of random diagrams, one representative from each class can be selected, allowing a more efficient assessment of tool performance and robustness while maintaining meaningful coverage.

The solution to this problem appears to lie in the idea of classification. However, the concept of a "class of UML class diagrams" is not well defined. The OMG UML specification does not formally introduce this notion. This ambiguity may be due in part to the fact that the concept of "class" lacks a single unified mathematical foundation. Various attempts have been made to incorporate the notion of class into set theory, but these often lead to well-known paradoxes. In this paper, we do not aim to engage in a deep philosophical exploration of these issues. Instead, we will propose a mathematical formalization of the problem, guided by intuitions drawn from the use of category theory in computer science [1].

The core idea behind our formal classification of UML class diagrams is the identification of basic templates, simple, parameterized class diagrams that can be used to describe more complex diagrams, which we call "complex templates". These complex structures are formed by multiplying (*i.e.*, using multiple instances of) and combining basic templates according to a set of defined combination rules. Each basic template corresponds to an instance of the minimal UML metamodel shown in **Figure 1**. A class diagram is classified as of a specific type if it is isomorphic to a constructed complex template. Conceptually, our approach aligns more closely with mereology—the theory of part-whole relationships [2] [3]—which shares foundational similarities with category theory.

We defined a number of basic templates based on the minimal UML metamodel shown in **Figure 1**. This UML metamodel is a simplification of the UML metamodel from [4] [5]. A UML metamodel is a UML diagram whose instantiations are all possible UML class diagrams. A metamodel includes Meta: Classes, Datatypes, Attributes, Associations, and Constraints. The minimal UML metamodel includes

Class, Association, Property, DataType and Generalization metaclasses. The minimal UML metamodel used in our experiments can be easily extended to support all types of UML class diagram elements, since all these types are part of the UML superstructure. Commonly used types of UML class diagram elements are class, class attribute, data type, binary association, unary association, N-ry association, operation, parameter, association, generalization, aggregation, composition, dependency, realization, interface, enumeration, enumeration literal and package.

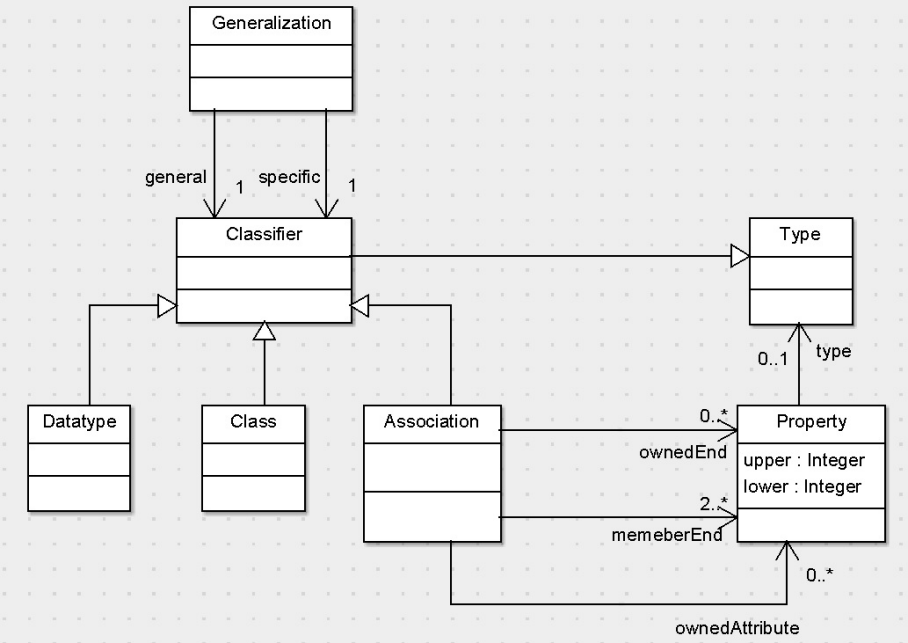


Figure 1. Minimal UML metamodel (adapted from [4] [5]).

The paper is organized as follows. Section 2 describes the formulation of the problem. Section 3 demonstrates an example of a class of class diagrams. Section 4 discusses related work. Section 5 describes the formalization of our method. Section 6 describes the method of classification of class diagrams. Section 7 gives details on basic templates. Section 9 describes the evaluation of the approach. Section 10 presents our conclusions.

2. Problem Formulation

In this section, we present the problem of classifying UML class diagrams into equivalence classes based on their structural properties. The central challenge is to develop a method for grouping class diagrams such that each group is represented by a canonical form, ensuring that diagrams within the same group are isomorphic and the classification covers all possible (or as many as possible) UML diagrams. To achieve this, we break down the problem into two subproblems. The first subproblem focuses on defining a set of basic templates and composition rules that allow any UML class diagram to be represented by a complex template, which serves as the canonical representative of the equivalence class containing

that diagram. This subproblem is solved manually as it requires careful selection of basic templates and determination of the principles and the rules of construction of complex templates. The second subproblem addresses the realization of this classification process by constructing the complex templates algorithmically and assigning the class diagrams to the appropriate equivalence classes. Together, these subproblems provide a structured approach to solving the broader classification problem. A first step towards the formalization of this problem is:

Problem (Classification of UML class diagrams) Develop a method to classify UML class diagrams into equivalence classes based on their structural properties.

Subproblem 1 (Representing classes of class diagrams) Define a set of basic templates, T, and composition rules, R, such that for any UML class diagram, D, a complex template, C, can be constructed using R s.t. D is isomorphic to C.

Subproblem 2 (Realization of classification of UML class diagrams) Develop an algorithm that solves the following problem: Given a UML class diagram, D, a set of basic templates, T, and a set of composition rules, R, construct a complex template, C, using a collection of basic templates, T, (possibly with repetitions) such that D is isomorphic to C.

3. Example of a Class of UML Class Diagrams

The following provides an example of a class of class diagrams. We begin with a textual description, followed by a UML class diagram representative of the class of class diagrams in the form of a complex template diagram. Also, we show how the complex template diagram is constructed from an intermediate complex template diagram and a copy of a basic template diagram. Lastly, we demonstrate how a specific class diagram is classified within this class of diagrams. The complex template diagram includes variables for classes, binary associations, association ends, attributes, and generalizations, with variable elements identified using the *var* stereotype.

Example: *Textual Description*

This class of class diagrams includes all class diagrams conforming to the following pattern. The diagram includes 5 classes interconnected with 2 generalizations, 4 directed associations and one bidirectional association in a particular way. One class in the diagram includes 2 attributes of String and Integer primitive types defined as data types. All association end multiplicities have unspecified range.

The diagram has the following properties. The average number of associations connected to a class is 2. Therefore, the diagram is not considered to be complex in terms of number of associations. The diagram exhibits one simple inheritance hierarchy. The structure of the diagram is asymmetric. There is at most one association between classes. Also, the diagram exhibits a small number of class attributes of different primitive types.

Complex template diagrams

Figure 4 shows a complex template diagram resulted from the composition of

an intermediate complex template diagram shown in Figure 2 and a copy of a basic template diagram shown in Figure 3. Initially, in the process of building the complex template in Figure 4, the copy of the basic template diagram has arbitrary names for the elements of the class diagram. Then, the class, association, and association end variables of the copy are renamed according to the composition rules in such a way that it shares two classes with the intermediate complex template diagram. However, the names of the association and the association ends in the copy are different from the names of the associations and association ends in the intermediate complex template diagram.

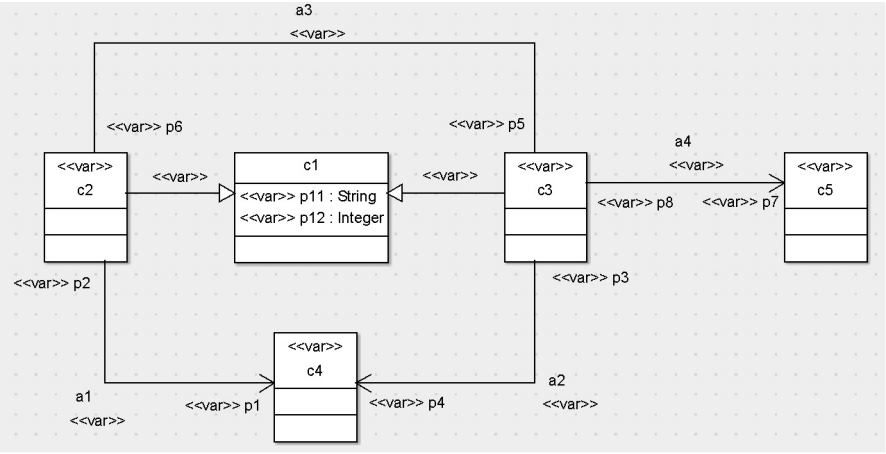


Figure 2. Intermediate complex template diagram.

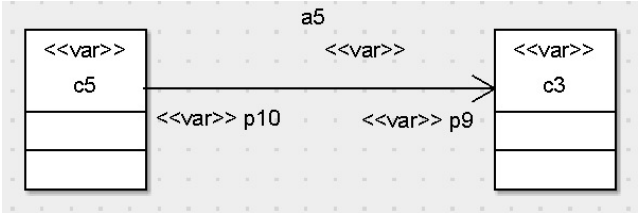


Figure 3. Basic template diagram copy.

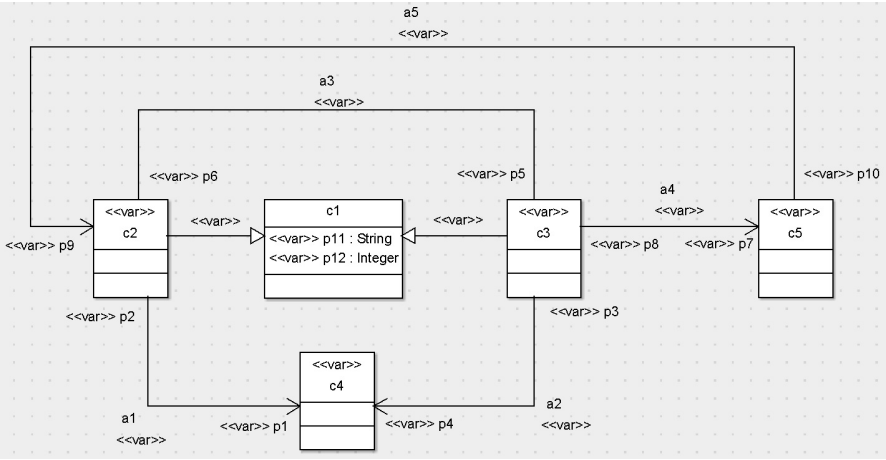


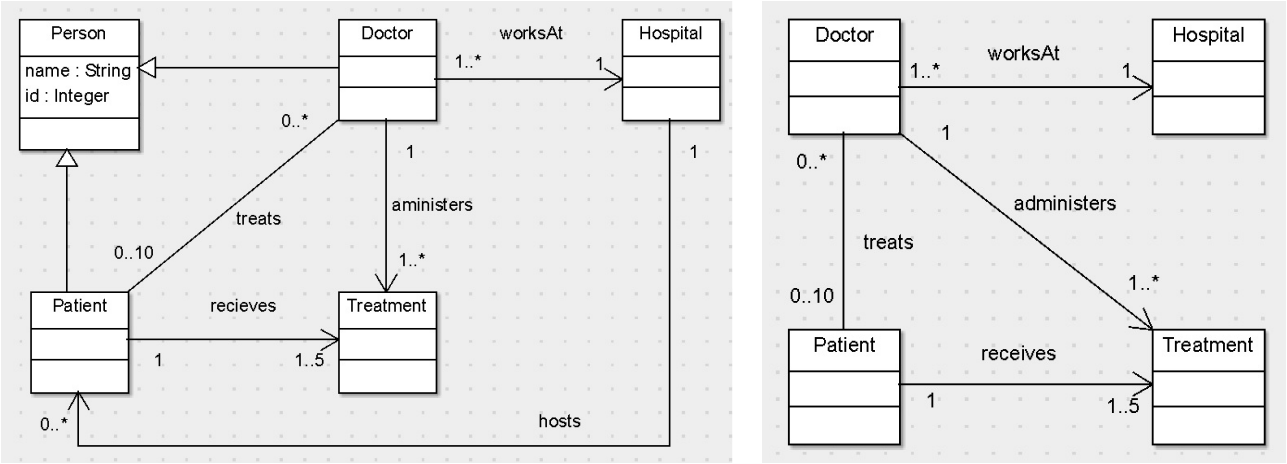
Figure 4. Complex template diagram.

The complex template diagram in Figure 4 is representative of the class of class diagrams described in the textual example above. Since UML class diagrams do not natively support variables for multiplicities, we are using 0..* multiplicity to give an unspecified range for association end variables. When the class of diagrams is instantiated, specific multiplicities are substituted in place of 0..*.

Classification of a class diagram

The class diagram in Figure 5(a) is classified into the class of class diagrams represented by the complex template in Figure 4 because it is isomorphic to this template (with the stereotypes removed). The two diagrams are structurally identical. The difference between the diagrams is the names of elements and their property values as well as different arrangements of classes.

The class diagram in Figure 5(b) cannot be classified into the class of class diagrams represented by this complex template, as it is not isomorphic with it. The diagrams are structurally different. The class diagram in Figure 5(b) includes four classes, three directed associations, and one bidirectional association, while the complex template diagram includes five classes, one bidirectional association, two class attributes of defined data type, two generalizations, four directed associations, and two data types.



(a) Healthcare diagram 1

(b) Healthcare diagram 2

Figure 5. Two class diagrams to be classified.

4. Related Work

4.1. Classification of Class Diagrams

We did not find extensive existing work about classification of class diagrams. There are some methods that use class diagram metrics and machine learning for classification of class diagrams. The method in [6] classifies class diagram into two classes: forward engineered class diagrams and reverse engineered class diagrams. The method in [7] predicts bad-smell for software design models. In particular, 7 bad-smells of the design are identified. The method in [8] predicts fault prone classes in class diagrams. These methods do not formally define classes of class

diagrams, identify very limited number of classes of class diagrams, and do not consider structure of class diagram during classification process.

4.2. Comparison of Class Diagrams

The method described in [9] compares class diagram elements—such as classes, attributes, operations, relationships, and class neighbors—based on the names of these elements. It employs individual similarity metrics for name-based comparisons and compound metrics that combine these individual metrics. The compound metrics are used for comparison of classes based on combination of names, attributes, operations and neighborhood information. Semantic similarity, including relationships like synonyms and hyponyms, is assessed using WordNet [10], which measures the distance between words in the WordNet hierarchy. However, this method does not account for the overall structure of class diagrams, which is its main limitation. Additionally, the accuracy of comparisons depends heavily on the weight assignment for each individual metric within the compound metric. The study calibrates these weights to evaluate the quality of class diagram comparisons. While the results indicate that compound metrics outperform individual metrics, they are based on only two use cases, limiting the generalizability of the findings.

The paper [11] discusses common types of overlaps between concepts in different models, addressing informal, semi-formal, and formal semantics. These overlaps are categorized as follows:

1. Equivalence: Two elements in different models are equivalent if they refer to the same real-world concept. For models with informal or semi-formal semantics, equivalence is determined based on human perceptions, beliefs, and assumptions. In contrast, for semantically formalized models, equivalence is established when elements share identical semantic properties. For example, in state machines, equivalence may depend on the associated behaviors of states.

2. Similarity: Two elements may exhibit partial semantic similarity without being semantically equivalent. Quantitative measures can be introduced to evaluate how closely the semantic properties of one element align with another.

3. Generalization: This overlap occurs when an element in one model serves as a generalization of elements in another model.

4. Aggregation: This overlap arises when two elements in different models are related through an aggregation relationship.

5. Overriding: One element in a source model may be disregarded in favor of a semantically similar element in another model.

6. Information Gaps: Information gaps between elements in different source models must be bridged to establish relationships between them. This often requires introducing abstract concepts and generalization relationships in the merged model.

However, this method lacks a systematic approach to identifying overlaps in class diagrams with informal or semi-formal semantics. Overlap identification is

performed manually, making it subjective and prone to variability based on individual interpretation.

The method described in [12] utilizes a 3-way merge approach to compare three versions of a class diagram. These versions include:

1. Developer Version: The diagram the developer wants to add to the repository.

2. Current Version: The most recent diagram version in the repository.

3. Base Version: The common ancestor from which both the developer and current versions are derived.

Each diagram is translated into Prolog facts, and rules based on the UML metamodel are applied to infer indirect relationships between diagram elements. The method then compares the enriched Prolog fact sets to determine if the Developer and Current versions of the diagrams are semantically equivalent, if one semantically contains the other, or there are conflicts. Semantic conflict detection involves performing two different operations: 1) Compare the base version with the developer version, identifying additions and deletions made by the developer. 2) Compares the base version with the current version, identifying changes made in the repository. This process requires all three diagram versions for comparison and relies on Prolog-based reasoning to detect semantic equivalence or conflicts.

The method [13] compares class diagrams based on ontology alignment and subgraph isomorphism. The comparison includes the following steps.

1. The class diagrams are translated to ontologies.

2. The mapping is created between diagram ontologies using similarity measures, background ontologies, and validation rules.

3. The isomorphic subgraphs are identified in the graphs representing the diagram ontologies using the mapping between the diagram ontologies.

This method does not ensure formal compliance with the metamodel. The validation rules are ontology oriented rather than class diagram oriented. Also, the semantics of class diagrams may be lost during their conversion to ontologies.

The methods in [14]-[16] utilize graph isomorphism [17] to find design patterns in class diagrams. These methods consider the situations where the design pattern fully or partially exists in the given diagram. The method in [14] compares the given class diagram and the design pattern diagram by comparing their labeled relationship graphs, where each node representing a class includes a label incorporating the number of superclasses, collaborating classes and subclasses for the class and each edge representing a relationship is labeled with relationship type. There is a support for dependency, generalization, direct association, and aggregation. Additionally, the method evaluates combinations of classes and their relationships across diagrams. The method in [15] extracts relationship graphs from the given class diagram and the design pattern diagram. There is a relationship graph for each type of relationship (e.g., association or generalization). Then, it compares corresponding relationship graphs for each type of relationship. The method [16] compares the given class diagram and the design pattern diagram by

comparing their labeled relationship graphs, where nodes represent classes and edges represent relationships between pairs of classes, with each edge assigned a weight that reflects the combination of multiple relationships (association, generalization, realization, or other/disconnected) between the two classes. The method computes a percentage of the partial existence of design pattern diagram in the given class diagram.

The method in [18] finds design patterns in class diagrams by performing graph comparisons based on heuristics. Similar to [15] it relies on extraction of relationship graphs for each type of relationship from the given class diagram and the design pattern diagram.

The method in [4] [5] compares class diagram elements to identify common elements for the class diagram composition. The classes and datatypes are compared by name, associations are compared based on association name, association end names, multiplicities and association end navigabilities, generalizations are compared based on general/specific class names and attributes are compared based on class name, attribute name and attribute type name. The method relies on formalization of UML metamodel. However, there is no structural comparison of class diagrams.

None of these methods performs structural comparison of class diagram and at the same time ensures formal compliance with the UML metamodel. Our method, on the other hand, compares class diagrams based on graph isomorphism and supports formal compliance with metamodel.

5. Formalization of the Classification of UML Class Diagrams

This section introduces the connection between UML class diagrams and category theory. It is captured by the mapping of class diagrams to E-graph objects of the category EGraphs [19], which represent the structure of the class diagrams. The relationships between class diagrams are morphisms of EGraphs. This is shown in Figure 6.

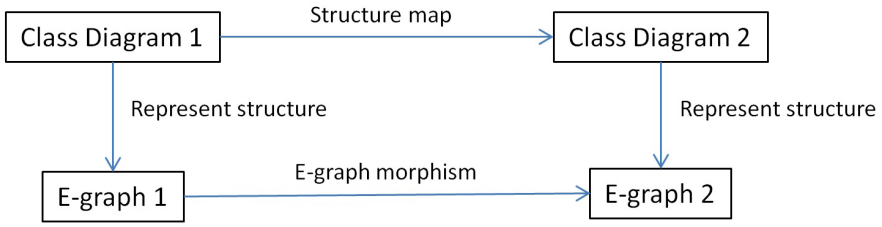


Figure 6. Mapping between structures of class diagrams.

Definition 1. Consider two E-graphs of basic templates B_1, B_2 and a common subgraph Q along with E-graph morphisms f_1, f_2 from Q to B_1 and B_2. The pushout object of such a diagram is a complex template. This will also apply to the situation where B_1 is a complex template and B_2 is a basic template.

Note: This definition does not cover all the aspects of complex template. The

completion of this construction requires the use of *ATGI*-graphs as discussed in Section 6.

In summary, Definition 1 provides a grouping of class diagrams into isomorphic classes using E-Graphs. The vertices and edges of these graphs do not explicitly include types. To achieve this, we represented the basic templates and UML class diagrams as attributed typed graphs respecting inheritance (*ATGI*-graph), as described in [4] [5] [19]. This graph allows us to capture the structure of class diagrams as well as types related to class diagrams. The minimal UML metamodel in **Figure 1** is represented as an attributed typed graph with inheritance (*ATGI*). The mapping between the E-graph representing the structure of the class diagram and the *ATGI* representing the minimal UML metamodel is represented using an *ATGI*-clan morphism. The notion of inheritance provides complexity reduction of graphs representing metamodels, as described in [19].

Figure 7 shows the graphical representation of the *ATGI* that captures the minimal UML metamodel. It shows the inheritance aspect (hollow arrows) and relationships (regular arrows) show the meta-associations, and the meta-attributes of the metamodel. This notation was borrowed from [19].

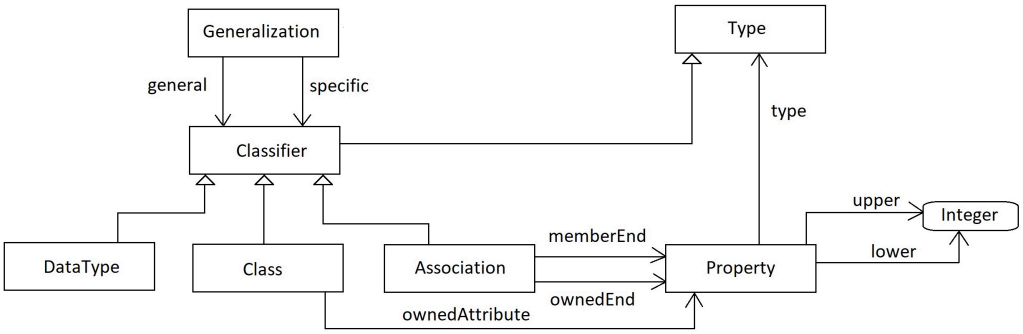


Figure 7. *ATGI* represents the minimal UML metamodel (adapted from [4] [5]).

The mapping between the E-graph representing the structure of a class diagram and the *ATGI* representing the minimal UML metamodel is represented using an *ATGI*-clan morphism.

In order to represent the structure of a class diagram and the typing of a class diagram by the minimal UML metamodel in a single graph, we introduce an inheritance-respecting typed attributed graph (*ATGI*-graph) [20].

Now we can return back to the main issue that we are trying to address, *i.e.*, the grouping of UML class diagrams into isomorphic classes. Towards this aim, we defined *complex templates* in Definition 1. As was stated earlier, the graphs used in that definition did not cover all of the aspects of UML class diagrams that we want to consider. So now we introduce the definition of an *equivalence class of UML Class Diagrams*. It will be defined by the following equivalence relation.

Definition 2. Consider a UML class diagram represented by an *ATGI*-graph, *D*. *D* is in the equivalence class [*C*] if *D* is isomorphic with the complex template *C* constructed according to Definition 1.

6. Algorithmic Classification of UML Class Diagrams

The formal framework described in the previous section will be used for the classification of UML class diagrams. The classification of class diagrams means the classification of class models represented by diagrams. The basic template diagrams are preprocessed so that the stereotypes are removed at the beginning of the classification process. The results, including intermediate results, and the steps of the classification process will be as follows.

1. Formalize the minimal UML metamodel as an attributed type graph with inheritance *ATGI.*

2. Represent all the basic templates as *ATGI*-graphs.

3. Represent a UML class diagram as an *ATGI*-graph.

4. Select a collection of basic templates, possibly with repetitions, and find morphisms from the collection to the *ATGI*-graph; the morphisms must cover all of the *ATGI*-graph.

5. Compute the complex template using the selected collection of basic templates according to Definition 1.

6. Search for a complex template P that is isomorphic to the generated complex template. If found, then classify given diagram as $[P]$; if not found, then add a new class for the generated complex template and classify the diagram as this class.

The tool implementation of the classification of UML class diagrams is available online [21].

6.1. Converting a UML Class Diagram to an *ATGI*-Graph

Each ArgoUML class diagram developed in ArgoUML studio is read using ArgoUML API for Java into ArgoUML model. ArgoUML model elements are Java objects. Each object represents a class diagram element. There are relationships between these objects identified by ArgoUML API methods. An AgroUML model is mapped to an *ATGI*-graph based on mapping of a UML model to an *ATGI*-graph as described in [4]. The structure of ArgoUML model is represented as E-graph. The typing of ArgoUML model by UML metamodel is represented using *ATGI*-clan morphism. The structure of a class diagram and the typing of a class diagram by the minimal UML metamodel are represented as *ATGI*-graph.

In order to compare E-graphs representing two diagrams for isomorphism, it is necessary to represent values of meta-attributes as unique V_Ds. If we have two class attributes with multiplicity of 1 in the diagram, there will be two copies of value 1 in the E-graphs. This is accommodated by defining each V_D as pair that consists of a value and a unique ID.

6.2. Mapping of the Basic Templates to a Diagram

The following describes the process of mapping of the basic templates to a diagram. The input of the process consists of an *ATGI*-graph $G^I = (G, type_G)$ representing the diagram, where G is an E-graph defined as $G = \left(G_{V_G}, G_{V_D}, G_{E_G}, G_{E_A}, \left(s_{G_i}, t_{G_i}\right)_{i\in\{G,A\}}\right)$ and $type_G : G \to ATGI$. $ATGI$ repre-

sents a UML metamodel shown in Figure 7. Additionally, the input includes a set B of *ATGI*-graphs representing basic templates. The output is the set BC^I that includes copies of graphs representing basic templates and injective E-graph morphisms f from the E-graph of each copy to G where morphisms f cover all of the G. The following are the steps of the process.

1. Find a subgraph S in G^I isomorphic to graph $B_i \in B$, such that E_G of S are not mapped to any previously created copy BC_k^I.

2. If such S is found then create a copy $BC_j^I = \left(BC_j, type_{BC_j}\right)$ of B_i, where $type_{BC_j} : BC_j \to ATGI$, and create an injective E-graph morphism $f_j : BC_j \to G$ that maps the elements of BC_j to the elements of the E-graph of S and go to step 1.

3. Repeat steps 1 and 2 for all the basic templates.

We have not found any efficient subgraph isomorphism algorithm that works with or can be easily adopted to *ATGI*-graphs and has ready to use tool implementation. In particular, we found the following subgraph isomorphism algorithm tools for typed attributed graphs. The Java based tool JGraphT [22] is based on VF2 subgraph isomorphism algorithm [23] and is efficient for sparse graphs. However, it does not support graphs with parallel edges where *ATGI*-graphs prepresenting UML class diagrams may have parallel graph edges in E_G. The GraMi (Graph Mining Library) tool [24] is efficient for sparse graphs. However, it is not a convenient choice for integration with our method tool implementation since it is a command line tool and not an Application Programming Interface (API). It requires the input graphs to be stored in a specific format in a file. The mapping can only be output in a console or a file. The Python based tools NetworkX [25] and graph-tool [26] are based on VF2 subgraph isomorphism algorithm and are efficient for sparse graphs. However, these tools do not enforce any particular type system for attributes for typed attributed graph. The types of vetices and edges are just attributes. The types of attributes themselves cannot come from some metamodel graph. Also, the attribute names do not have types.

To this extent, we developed algorithms for finding a subgraph in the class diagram *ATGI*-graph that is isomorphic to a basic template *ATGI*-graph. The algorithms are based on Depth First Search (DSF) to explore candidate nodes and edges, backtracking (reversing the mapping decisions to recover from invalid paths in the search tree), and pruning (degree filtering) similar to the existing subgraph isomorphism algorithms VF2 algorithm and GraMi.

The algorithms guarantee to find isomorphic subgraphs for the basic template diagrams of the following types based on the minimal UML metamodel and are efficient for them.

- Basic template diagrams that consist of two classes and a binary relationship between them (e.g. bidirectional association).
- Basic template diagrams that consist of N-classes and an N-ry association between them.

- Basic template diagrams that contain one class with an attribute whose type is another class.
- Basic template diagrams that contain a class with an attribute of a specific primitive type defined as a data type.
- Basic template diagrams that contain a class with an attribute of a defined data type.

These types of basic template diagrams consist of minimal number class diagram elements and cover all types of class diagram elements supported by the minimal UML metamodel. The algorithms are applied to four specific basic templates, introduced in Section 7, which fall within the supported types above. The algorithms can be easily extended to support all types of UML class diagram elements. This is explained in the detailed description of the algorithms.

The pseudocode for the algorithms is shown in Algorithm 1 and 2. For simplicity of presentation, the algorithms exclude processing of V_Ds and E_As. The matching of V_Ds and E_As is done based on types. The backtracking of mappings for V_Ds and E_As is performed with respect to mismatched V_Gs.

The algorithms use the following notations.

1. *getNodeList* (*graph*): A method that returns a list of V_Gs of the *ATGI*-graph.

2. *getNodeListByType* (*graph*, *nodeType*): A method that returns a list of V_Gs of the *ATGI*-graph of a given type. The type is specified as a string value (e.g. "Class").

3. *getEdgeList* (*graph*): A method that returns a list of E_Gs of the *ATGI*-graph.

4. *source* (*edge*): A method that returns the source of an *ATGI*-graph graph edge.

5. *target* (*edge*): A method that returns the target of an *ATGI*-graph graph edge.

6. *getOutgoingEdgeList* (*node*): A method that returns a list of outgoing graph edges of the given *ATGI*-graph graph vertex.

7. *getIncomingEdgeList* (*node*): A method that returns a list of incoming graph edges of the given *ATGI*-graph graph vertex.

8. *nodeMapping*: A stack of pairs with a graph vertex in V_G of the basic template *ATGI*-graph as a first element and graph vertex in V_G of the diagram *ATGI*-graph a second element. This stack represents a mapping between graph vertices of the basic template and diagram *ATGI*-graphs.

9. *edgeMapping*: A stack of pairs with a graph edge in E_G of the basic template *ATGI*-graph as a first element and graph edge in E_G of the diagram *ATGI*-graph a second element. This stack represents a mapping between graph edges of the basic template and diagram *ATGI*-graphs.

10. *nodeMatchCount*: A global variable that counts matches of *ATGI*-graph graph vertices.

11. *setMatchId* (*node*): A method that sets a unique auto-incremented match ID for an *ATGI*-graph graph vertex. When there is a match between a graph vertex in V_G of the basic template *ATGI*-graph and a graph vertex in V_G of the diagram *ATGI*-graph, both graph vertices are given a unique auto-incremented in-

teger match ID. The default value of match ID is 0.

12. *getMatchId* (*node*): A method that return the match ID of an *ATGI*-graph graph vertex.

13. *setMatched* (*edge*): A method that set a match indicator of an *ATGI*-graph graph edge to true or false. When there is a match between a graph edge in E_G of the basic template *ATGI*-graph and a graph edge in E_G of the diagram *ATGI*-graph, the match indicator for both graph edges is set to true. The default value of match indicator is false.

14. *isMatched* (*edge*): A method that return the match indicator of an *ATGI*-graph graph edge.

15. *getType* (*node*): A method that returns the type of an *ATGI*-graph graph vertex as as string value.

16. *getType* (*edge*): A method that returns the type of an *ATGI*-graph graph edge as as string value.

17. *backtrackNodeMatchIds* (*nodeMapping*,*mismatchedNode*): A method that sets to 0 the match IDs of the *ATGI*-graph graph vertices in all the pairs from the top of the stack representing the mapping between graph vertices of the basic template and diagram *ATGI*-graphs up to and including the pair that includes the given mismatched graph vertex of the basic template *ATGI*-graph.

18. *backtrackEdgeMatchFlags* (*edgeMapping*,*mismatchedEdge*): A method that sets to false the match indicators of the *ATGI*-graph graph edges in all the pairs from the top of the stack representing the mapping between graph edges of the basic template and diagram *ATGI*-graphs up to and including the pair that includes the given mismatched graph edge of the basic template *ATGI*-graph.

19. *backtrackEdgeMapping* (*edgeMapping*,*mismatchedEdge*): A method that pops all the *ATGI*-graph graph edge pairs from the stack representing the mapping between graph vertices of the basic template and diagram *ATGI*-graphs up to and including the pair that includes the given mismatched graph edge of the basic template *ATGI*-graph.

20. *backtrackNodeMapping* (*nodeMapping*,*mismatchedNode*): A method that pops all the *ATGI*-graph graph vertex pairs from the stack representing the mapping between graph edges of the basic template and diagram *ATGI*-graphs up to and including the pair that includes the given mismatched graph vertex of the basic template *ATGI*-graph.

The Algorithm 1 takes as input a diagram *ATGI*-graph and a basic template *ATGI*-graph. It outputs true if a subgraph in the diagram *ATGI*-graph is found that is isomorphic to the basic template *ATGI*-graph and false otherwise. Also, it takes by reference empty mappings between graph vertices and graph edges of the basic template and diagram *ATGI*-graphs. Pairs of *ATGI*-graph graph vertices and graph edges are added to and removed from the mappings during the exploration of isomorphic subgraphs.

First, the algorithm retrieves the lists of graph vertices of Class type from the

diagram *ATGI*-graph and basic template *ATGI*-graph. Then, it tries to explore the matching subgraphs in the diagram *ATGI*-graph starting at each graph vertex of Class type. This is done by calling Algorithm 2. The graph vertex from which the basic template *ATGI*-graph traversal starts is always the first graph vertex in the list of graph vertices of Class type in the basic template *ATGI*-graph. Also, the following is done before each call of Algorithm 2. The mappings between graph vertices and graph edges of the basic template and diagram *ATGI*-graphs are cleared. Each graph vertex of Class type of the diagram *ATGI*-graph is considered to have a candidacy match with the identified graph vertex of Class type of the basic template *ATGI*-graph. Therefore, a corresponding pair of graph vertices is added to the mapping between graph vertices of the basic template and diagram *ATGI*-graphs. The algorithm can be extended to support *ATGI*-graph vertices representing interfaces, packages or stereotypes as starting points for matching subgraph exploration.

Algorithm 1: FindSubgraphIsomorphizm

Input: *diagramGraph* - diagram ATGI-graph; *basicTemplateGraph* - basic template ATGI-graph
InputByRef: *nodeMapping* - mapping between graph vertices of the basic template and diagram ATGI-graphs; *edgeMapping* - mapping between graph edges of the basic template and diagram ATGI-graphs
Output: *true* if isomorphic subgraph is found in the diagram ATGI-graph; *false* otherwise

```
1  diagramClassNodeList ← getNodeListByType(diagramGraph,"Class")
2  basicTemplateClassNodeList ← getNodeListByType(basicTemplateGraph,"Class")
3  isomorphizmFound ← false
4  foreach classNode ∈ diagramClassNodeList do
5      foreach node ∈ getNodeList(diagramGraph) do
6          setMatchId(node) ← 0
7      foreach edge ∈ getEdgeList(diagramGraph) do
8          setMatched(edge) ← false
9      nodeMapping.clear(), edgeMapping.clear()
10     nodeMatchCount ← 1
11     setMatchId(classNode) ← 1, setMatchId(basicTemplateClassNodeList[0]) ← 1
12     nodeMapping.push(⟨basicTemplateClassNodeList[0], classNode⟩)
13     if exploreCandidateSubgraphs(classNode, basicTemplateClassNodeList[0], nodeMapping, edgeMapping) then
14         isomorphizmFound ← true
15         break
16 return isomorphizmFound
```

The Algorithm 2 takes as input *diagramNode*, a graph vertex from the diagram *ATGI*-graph, and *basicTemplateNode*, a graph vertex from the basic template *ATGI*-graph. It outputs true if a subgraph in the diagram *ATGI*-graph is found that is isomorphic to a subgraph in the basic template *ATGI*-graph by starting traversal from *basicTemplateNode* in the basic template *ATGI*-graph and staring exploration from *diagramNode* in the diagram *ATGI*-graph. The output is false if no subgraph isomorphism found. Also, it takes by reference empty mappings between graph vertices and graph edges of the basic template and diagram *ATGI*-graphs.

Algorithm 2: ExploreCandidateSubgraphs

Input: *diagramNode* - graph vertex of the diagram ATGI-graph; *basicTemplateNode* - graph vertex of the basic template ATGI-graph
InputByRef: *nodeMapping* - mapping between graph vertices of the basic template and diagram ATGI-graphs; *edgeMapping* - mapping between graph edges of the basic template and diagram ATGI-graphs
Output: *true* if isomorphic subgraph is found in the diagram ATGI-graph; *false* otherwise

```
1  basicTemplateOutgoingEdgeList ← getOutgoingEdgeList(basicTemplateNode)
2  diagramOutgoingEdgeList ← getOutgoingEdgeList(diagramNode)
3  basicTemplateIncomingEdgeList ← getIncomingEdgeList(basicTemplateNode)
4  diagramIncomingEdgeList ← getIncomingEdgeList(diagramNode)
5  if getType(basicTemplateNode) ∈ ["Association","Generalization","Property"] then
6      if |basicTemplateOutgoingEdgeList| ≠ |diagramOutgoingEdgeList| then return false
7      if |basicTemplateIncomingEdgeList| ≠ |diagramIncomingEdgeList| then return false
8  else
9      if |basicTemplateOutgoingEdgeList| > |diagramOutgoingEdgeList| then return false
10     if |basicTemplateIncominggEdgeList| > |diagramIncomingEdgeList| then return false
11 basicTemplateEdges ← [basicTemplateOutgoingEdgeList, basicTemplateIncomingEdgeList]
12 diagramEdges ← [diagramOutgoingEdgeList, diagramIncomingEdgeList]
13 for i ∈ (0, 1) do
14     foreach edge1 ∈ basicTemplateEdges[i] do
15         nodeMatchFound ← false
16         if isMatched(edge1) then continue
17         foreach edge2 ∈ diagramEdges[i] do
18             if isMatched(edge2) then continue
19             node1 ← i = 0 ? target(edge1) : source(edge1)
20             node2 ← i = 0 ? target(edge2) : source(edge2)
21             if getType(edge1) = getType(edge2) ∧ getType(node1) = getType(node2) ∧ getMatchId(node1) =
                 getMatchId(node2) then
22                 if getMatchId(node1) = 0 ∧ getMatchId(node2) = 0 then
23                     nodeMatchCount ← nodeMatchCount + 1
24                     setMatchId(node1) ← nodeMatchCount, setMatchId(node2) ← nodeMatchCount
25                     setMatched(edge1) ← true, setMatched(edge2) ← true
26                     nodeMapping.push(⟨node1, node2⟩)
27                     edgeMapping.push(⟨edge1, edge2⟩)
28                     if exploreCandidateSubgraphs(node2, node1, nodeMapping, edgeMapping) then
29                         nodeMatchFound ← true
30                         break
31                     else
32                         backtrackNodeMatchIds(nodeMapping, node1)
33                         backtrackEdgeMatchFlags(edgeMapping, edge1)
34                         backtrackNodeMapping(nodeMapping, node1)
35                         backtrackEdgeMapping(edgeMapping, edge1)
36                 else if getMatchId(node1) ≠ 0 ∧ getMatchId(node2) ≠ 0 then
37                     setMatched(edge1) ← true, setMatched(edge2) ← true
38                     edgeMapping.push(⟨edge1, edge2⟩)
39                     nodeMatchFound ← true
40                     break
41         if !nodeMatchFound then return false
42 return true
```

The algorithm recursively traverses through the basic template *ATGI*-graph starting from *basicTemplateNode* and tries to find a matching subgraph in the diagram *ATGI*-graph. The search for a matching subgraph in the diagram *ATGI*-graph starts from *diagramNode*. A neighbor graph vertex *node*1 of *basicTemplateNode* matches a candidate neighbor graph vertex *node*2 of *diagramNode* if *node*1 and *node*2 have the same type, the graph edge between *basicTemplateNode*

and *node*1 and graph edge between *diagramNode* and *node*2 have the same type, and the match IDs of *node*1 and *node*2 are the same. The match IDs can be both 0 in case where *node*1 and *node*2 were not matched earlier or some integer values greater than 0 in case where *node*1 and *node*2 were previously matched for candidacy. If the match IDs for *node*1 and *node*2 are both 0, a pair of *node*1 and *node*2 and a pair of corresponding graph edges are added to the mappings between graph vertices and graph edges of the basic template and diagram *ATGI*-graphs, and a recursive call to Algorithm 2 is made with *node*1 and *node*2 as parameters. If the recursive call to Algorithm 2 does not find a subgraph isomorphism, *node*1 and *node*2 are considered mismatched, and the mappings between the graph vertices and graph edges of the basic template and diagram *ATGI*-graphs are backtracked based on the mismatched *node*1 and its corresponding graph edge. If the match IDs for *node*1 and *node*2 are some integer values greater than 0, a pair of corresponding graph edges are added to the mapping between graph edges of the basic template and diagram *ATGI*-graphs. The algorithm can be extended to support operation, parameter, dependency, realization, enumeration literal, association class, generalization set, expression for termination condition on line 5. The elements of these types should match exactly between the diagram and basic template with respect to all connected meta-associations. However, the comparison on lines 6 and 7 should exclude meta-associations connected to stereotypes, comments and constraints.

6.3. Construction of a Complex Template

The common elements of copies of graphs representing basic templates are determined using morphisms from these copies to the graph representing the class diagram. Since the process of construction of a graph representing a complex template relies on intermediate complex templates as introduced in Section 5, it is necessary to know the common elements of graphs representing intermediate complex templates and copies of basic pattern graphs. In order to achieve this, the V_Gs of the copies of the graphs representing the basic templates are renamed in the following way. V_Gs in each copy are given unique names in such a way that two V_Gs in two copies that are mapped to the same graph vertex in V_G of the graph representing the diagram are given the same name. All E_Gs, V_Ds, E_As, sources and targets in each copy are updated accordingly.

The following describes the process of construction a graph representing a complex template from a collection of copies of graphs representing the basic templates. This process is based on the computation of a shared union of a pair of *ATGI*-graphs in [4] [5]. The input is the set BC^I that includes copies of *ATGI*-graphs representing the basic templates. The output is an *ATGI*-graph representing a complex template composed from all the copies in BC^I. The step of the process are as follows.

1. Pick two copies $BC_i^I = \left(BC_i, type_{BC_i}\right)$, $BC_j^I = \left(BC_j, type_{BC_j}\right)$ of the *ATGI*-graphs representing the basic templates, where $type_{BC_i} : BC_i \to ATGI$ and

$type_{BC_j} : BC_j \to ATGI$.

2. Construct the disjoint unions of the components (V_G, E_G, V_D, E_A) of BC_i and BC_j.

3. Define equivalence relations on the disjoint unions from the previous step and insert elements of the disjoint unions to P (representing the structure of the complex pattern) where equivalent elements are glued together.

4. Construct injective E-graph morphisms $h_i : BC_i \to P$, $h_j : BC_j \to P$ using the equivalence relations defined in the previous step.

5. Compute an E-graph Q (representing the largest common subgraph of BC_i and BC_j) and E-graph morphisms $g_i : Q \to BC_i$, $g_j : Q \to BC_j$ as the pullback using BC_i, BC_j, P, h_i and h_j.

6. Compute an *ATGI*-graph $P^I = \left(P, type_P\right)$ representing the complex pattern, where $type_P : P \to ATGI$, using $type_{BC_i}$, $type_{BC_j}$ and the pushout $\left(P, h_i, h_j\right)$.

7. Repeat steps 1 - 6 starting from the composition of P^I and the next copy BC_k^I and continuing for all remaining copies.

Steps 1-6 of the above process are shown in **Figure 8**.

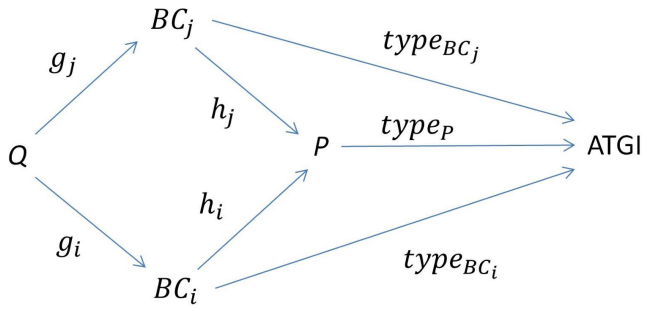


Figure 8. Construction of a complex template.

The details of computing steps 2 - 6 are shown in [5]. All elements of P are renamed to the representatives of the equivalence classes. Sources and targets as well as h_i and h_j are updated accordingly.

The time complexity of the algorithm of construction of a complex template is $O\left(n^2 * m\right)$, where n is the total of number of classes, associations, generalizations and class attributes in the class diagram and m is number of copies of basic templates used for the composition of the complex template.

6.4. Comparison of *ATGI*-Graphs

The complex template *ATGI*-graph and the given class diagram *ATGI*-graph are converted to colored multigraphs, where V_Gs, E_Gs are vertices and sources and targets are edges. Each type of V_Gs and E_Gs corresponds to a color. We omit E_{NA}s and V_Ds from this conversion for performance improvement since class diagram elements of each type have the same set of attributes based on the given UML

metamodel.

The outline of the algorithm for generating a colored multigrah graph from an *ATGI*-graph is the following. It takes an *ATGI*-graph as input and outputs a colored multigraph graph based on V_Gs and E_Gs.

1. Map all the types of V_Gs and E_Gs of the *ATGI*-graph to the unique colors (represented by integer values).

2. For each graph vertex v in V_G of the *ATGI*-graph, create a colored graph vertex with color corresponding to the type of v.

3. For each graph edge e in E_G of the *ATGI*-graph.

(a) Create a colored graph vertex c_1 with a color corresponding to the type of e.

(b) Find an existing colored graph vertex c_2 corresponding to the source of e.

(c) Find an existing colored graph vertex c_3 corresponding to the target of e.

(d) Create an undirected edge in the colored graph between c_2 and c_1.

(e) Create an undirected edge in the colored graph between c_1 and c_3.

After that, two colored multigraphs are compared for isomorphism using *jbliss* tool [27] based on [28] [29]. The algorithm guarantees finding isomorphic graphs. This algorithm is efficient for sparse graphs. In practice, *ATGI*-graphs representing class diagrams tend to be sparse. This is due to modular design, design patterns, hierarchical structure, and limited relationships between classes. The advantage of this tool is that it can be easily integrated with Java used to implement the diagram classification algorithm. Other similar algorithms cannot be easily integrated with Java.

7. Basic Templates

In this section, we introduce basic templates that are sufficient for classification of any class diagram that is an instance of the minimal UML metamodel and consists of class diagram elements of types covered by the basic templates. The basic templates are constructed in such a way that they consist of minimal number class diagram elements and cover most of class diagram element types supported by the minimal UML metamodel. In particular, they support the following class diagram concepts: class, generalization, binary association, association end, association end multiplicity, association end navigable property, data type (for representing primitive data types), attribute, and attribute type. The basic template diagrams include variables for classes, binary associations, association ends, attributes, data types, and generalizations. The variable elements are identified using the *var* stereotype.

To provide for full coverage of UML Class diagrams, the set of basic templates would need to be extended to cover all types of UML class diagram elements by following the principle of minimality of number of class diagram elements in each basic template as demonstrated with the minimal UML metamodel.

ATGI-graph representation of each basic template uses a compact notation

[19]. Each vertex in V_G and V_D, and edge in E_G and E_A has a type defined by *ATGI* representing the minimal UML metamodel. The types for each vertex in V_G use UML-like notation. For edges in E_G and E_A, only types are specified. For vertices in V_D, types are omitted. The elements of the basic template diagrams are mapped to graph vertices V_G. The meta-associations between the elements of a basic template diagram are mapped to graph edges E_G. The meta-attributes of the elements in a basic template diagram are mapped to node attribute edges E_A, and the values of the meta-attributes are mapped to data vertices V_D.

7.1. Template 1

This basic template diagram represents a class of class diagrams that includes all class diagrams that conform the following pattern. The diagram has a class *c*1 and a datatype *dt*1, where class *c*1 has a property *p*1 of type *dt*1. The basic template diagram is shown in Figure 9.

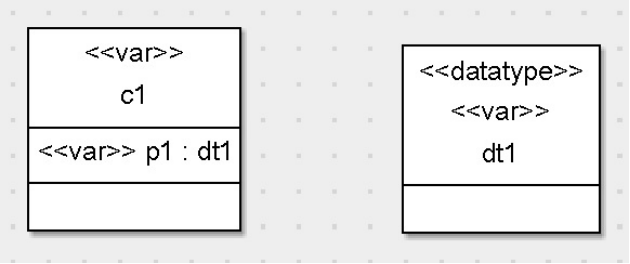


Figure 9. Template 1 diagram.

Figure 10 shows *ATGI*-graph representation of this template. Here, we have the following graph vertices V_G.

1. *dt*1: *Datatype* represents datatype *dt*1.
2. *c*1: *Class* represents class *c*1.
3. *p*1: *Property* represents class attribute *p*1.

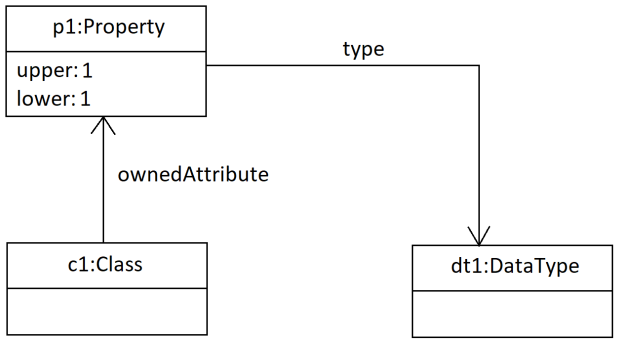


Figure 10. Typed attributed graph representation of template 1. The graph is shown in compact notation.

7.2. Template 2

This basic template diagram represents a class of class diagrams that includes all

class diagrams that conform the following pattern. The diagram has classes *c*1 and *c*2, and directed association *a*1 from *c*1 to *c*2. The association has a navigable end *p*1 on *c*2 with an unspecified multiplicity. The association has a non-navigable end *p*2 on *c*1 with an unspecified multiplicity. The basic template diagram is shown in Figure 11.

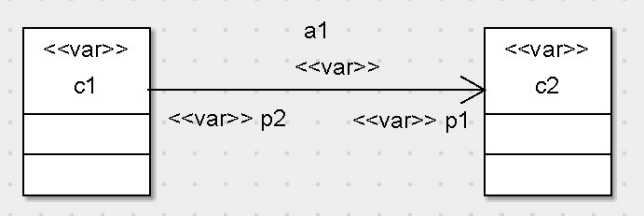


Figure 11. Template 2 diagram.

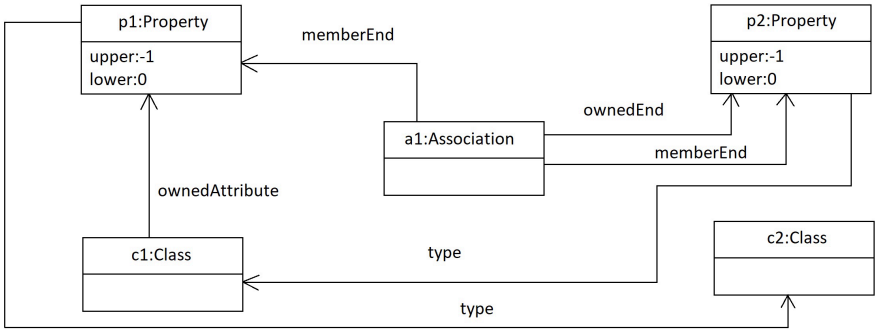


Figure 12. Typed attributed graph representation of template 2. The graph is shown in compact notation.

Figure 12 shows the *ATGI*-graph representation of this template. Here, we have the following graph vertices V_G.

1. *c*1: *Class*, *c*2: *Class* represent classes *c*1 and *c*2 respectively.
2. *a*1: *Association* represents association *a*1.
3. *p*1: *Property* and *p*2: *Property* represent association ends *p*1 and *p*2 respectively.

7.3. Template 3

Figure 13. Template 3 diagram.

This basic template diagram represents a class of class diagrams that includes all class diagrams that conform to the following pattern. The diagram has classes

*c*1 and *c*2, and a generalization relationship where *c*1 is subclass of *c*2. The basic template diagram is shown in Figure 13.

Figure 14 shows the *ATGI*-graph representation of this template. Here, we have the following graph vertices V_G.

1. *c*1: *Class*, *c*2: *Class* represent classes *c*1 and *c*2 respectively.
2. *g*1: *Generalization* represents generalization between classes *c*1 and *c*2.

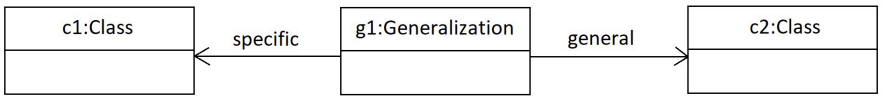


Figure 14. Typed attributed graph representation of template 3. The graph is shown in compact notation.

7.4. Template 4

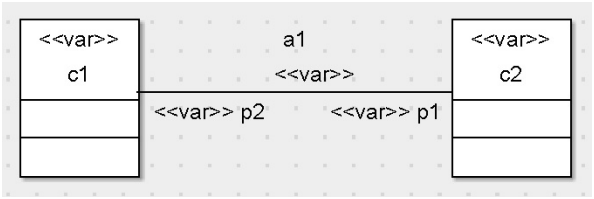


Figure 15. Template 4 diagram.

This basic template diagram represents a class of class diagrams that includes all class diagrams that conform to the following pattern. The diagram has classes *c*1 and *c*2, and a bidirectional association *a*1 between *c*1 and *c*2. The association has navigable ends *p*1 and *p*2 on *c*2 and *c*1 respectively with unspecified multiplicities. The basic template diagram is shown in Figure 15.

The *ATGI*-graph representation is similar to the representation of basic template 2. In this case, *a*1:*Association* node is connected to *p*1:*Property* and *p*2:*Property* nodes via two *ownedEnd* edges. Also, there is no *ownedAttribute* edge from *c*1:*Class* node to *p*1:*Property* node.

8. Classification of Class Diagram Use Case

8.1. A Class Diagram Coverage

We begin by showing how the basic templates are mapped to the structure of the class diagram under analysis. Figure 16 shows a copy $BC_1^I = \left(BC_1, type_{BC_1}\right)$ of an *ATGI*-graph representing the basic template 3 shown in Figure 13, a copy $BC_2^I = \left(BC_2, type_{BC_2}\right)$ of an *ATGI*-graph representing the basic template 2 shown in Figure 11, where $type_{BC_i} : BC_i \to ATGI$ for $i = \{1,2\}$. The diagram to be classified shown in Figure 17 is represented as *ATGI*-graph $G^I = \left(G, type_G\right)$, where $type_G : G \to ATGI$. The E-graph morphisms $f_1 : BC_1 \to G$ and $f_2 : BC_2 \to G$ are injective and jointly surjective. For the simplicity of representation, we flattened the internal multicomponent structure of E-graph morphisms f_1 and f_2.

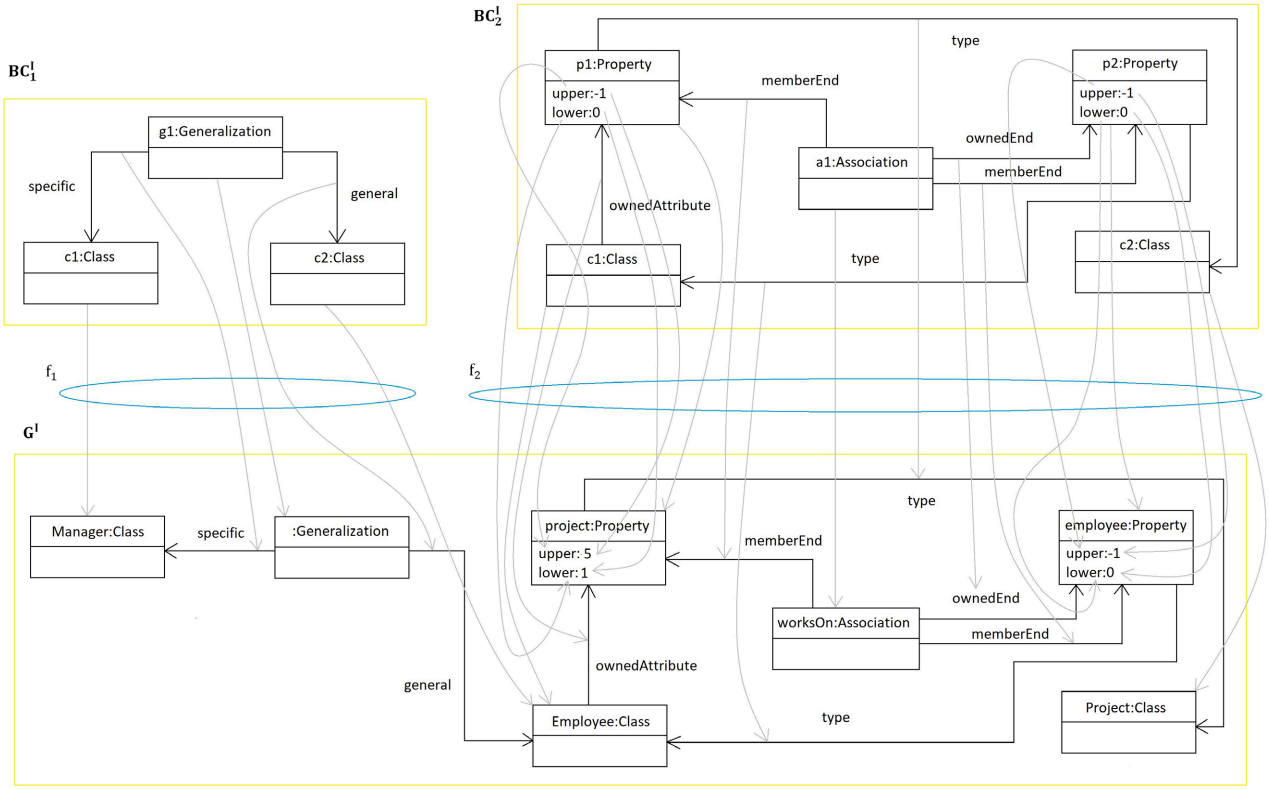


Figure 16. Mappings from the graphs of copies of basic templates to the graph of a class diagram.

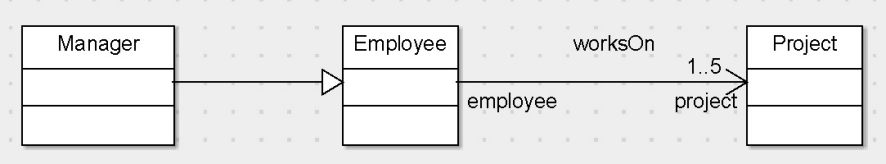


Figure 17. Class diagram to be mapped to the basic templates.

Each vertex V_G of G^I is mapped to one vertex V_G in BC_1^I or BC_2^I, or two vertices V_G in BC_1^I and BC_2^I. Each edge E_G of G^I is mapped to one edge E_G in BC_1^I or BC_2^I. Therefore, G^I is fully covered by BC_1^I and BC_2^I.

8.2. A Complex Template Construction

In this section, we demonstrate the composition of copies of graphs representing basic templates into a graph representing a complex template.

Figure 18 shows copies $BC_1^I = \left(BC_1, type_{BC_1}\right)$ and $BC_2^I = \left(BC_2, type_{BC_2}\right)$ of an *ATGI*-graph representing the basic template 3 shown in Figure 13 and an *ATGI*-graph representing the basic template 2 shown in Figure 11, respectively, where $type_{BC_i} : BC_i \to ATGI$ for $i = \{1,2\}$. The E-graph Q represents common subgraph of BC_1 and BC_2. The *ATGI*-graph $P^I = \left(P, type_P\right)$ representing a complex template, where $type_P : P \to ATGI$. The E-graph morphisms

$g_1 : Q \to BC_1$, $g_2 : Q \to BC_2$, $h_1 : BC_1 \to P$ and $h_2 : BC_2 \to P$ are injective.

Copies BC_1^I and BC_2^I are renamed as described in Section 6.3. For example, $c1$ in BC_1^I and $c2$ in BC_2^I as shown in Figure 16 are renamed to $c2$ since they are both mapped to *Employee* in G^I. The construction of P^I is performed according to the steps in Section 6.3.

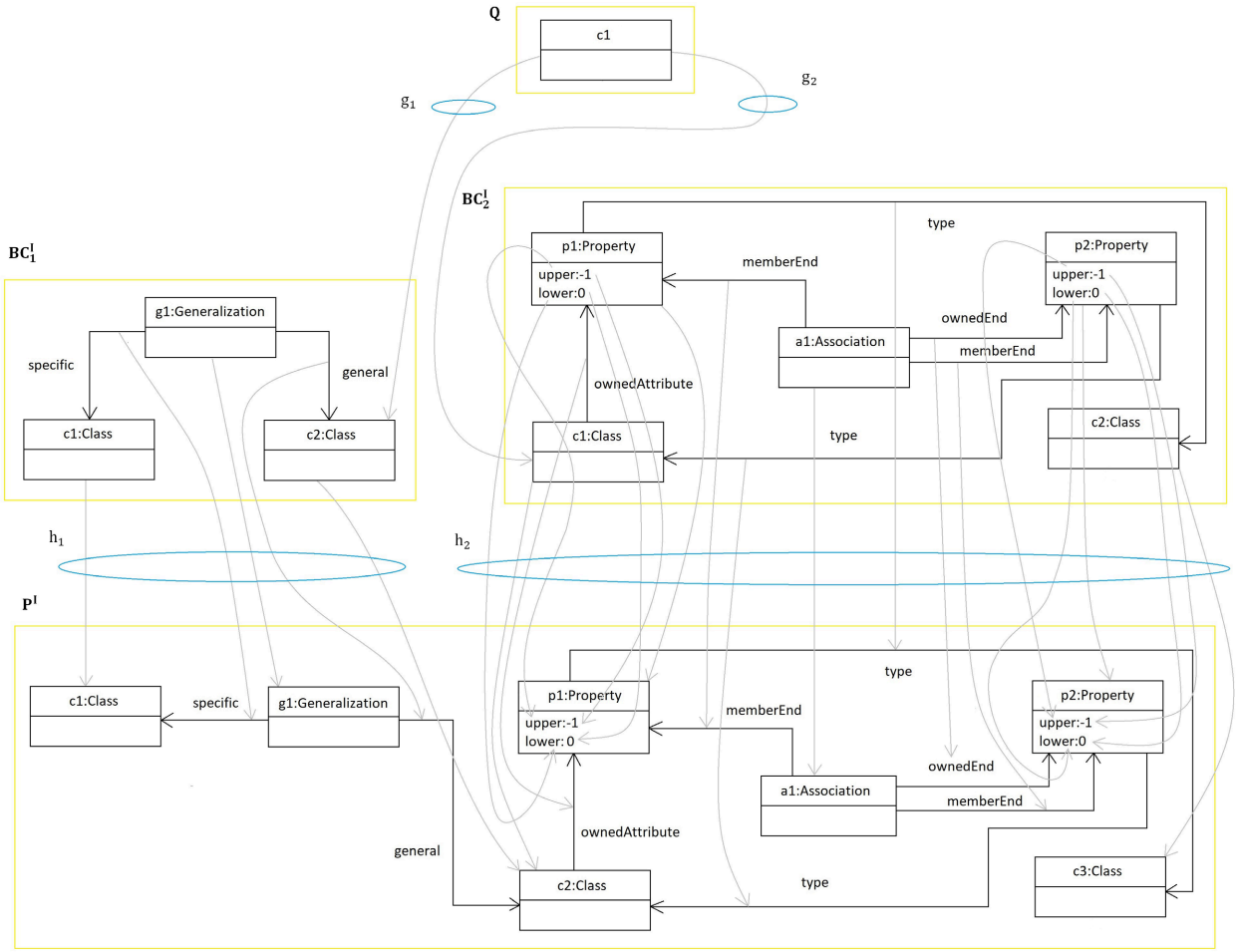


Figure 18. Construction of a complex template.

8.3. Classification of a Class Diagram

The *ATGI*-graph P^I and G^I are isomorphic since they are structurally identical. The class diagram in Figure 17 is classified into a class of diagrams represented by the complex template P^I.

9. Evaluation of the Diagram Classification

Our classification method was evaluated with respect to the preservation of the structure and typing of basic templates during composition, the equivalence of class diagrams and complex templates, accuracy and efficiency.

The structure of the basic templates during the construction of a complex template is preserved by the colimit. This is discussed in [11]. The preservation of

typing for the basic templates with respect to the minimal UML metamodel during composition is ensured by the composition of *ATGI*-graphs using the colimit. This is shown in [19].

The equivalence of a class diagram and a complex template with respect to the structure and typing of elements is ensured by the isomorphism of the *ATGI*-graphs representing the class diagram and the complex template.

It is necessary to show that any class diagram based on the given minimal UML metamodel can be grouped into isomorphic classes using a set of basic templates by creating (possibly) multiple copies of the basic templates and combining them according to the combination rules. To achieve this, we conducted an experiment that includes the following steps.

1. Generate random diagrams that are instances of the minimal UML metamodel and consist of class diagram elements of types covered by the basic templates.

2. For each random diagram, construct a complex template by composing copies of the basic templates.

3. If there is an isomorphism between the random diagram and the corresponding complex template then this random diagram is assigned to a class represented by the complex template.

In this experiment, the input basic templates and the UML class diagram were developed in the open-source ArgoUML studio and mapped to *ATGI*-graphs. The number of random diagrams for which isomorphisms with the complex templates were found were counted. The ratio of classified to all generated random diagrams serves as the measure of coverage of the test space, *i.e.*, the space of types of UML diagrams. Also, the number of isomorphic classes of diagrams is counted.

We did not find any large spaces of industrial class diagrams. Therefore, we conducted diagram classification experiments on two kinds of diagram spaces that simulate a common structure of real-world diagrams and are efficient for testing with respect to structural diversity and size. The spaces are based on patterns observed in the standard modeling documentation and practice. A space is efficient for testing if it includes a significant number of varied diagram characteristics while minimizing trivial variations, *i.e.* changes that do not meaningfully affect the overall diagram structure. Spaces dominated by trivial variations tend to be unnecessarily large and less informative. The diagrams in the constructed spaces have arbitrary element names (e.g. $c1$, $c2$, …, cn for classes and $a1$, $a2$, …, an for associations). The diagram structure varies over an initial set of classes with given names. The following steps outline a general procedure for constructing synthetic spaces of UML class diagrams that reflect commonly occurring structures in practice. We applied this procedure to define the two specific spaces used in our evaluation.

1. Decide the size of diagrams in terms of number of classes.

2. Decide a level of design (e.g. Domain-Driven Design [30], Logical Design [31], or Detailed Design [31]).

3. Decide a type of system (e.g. system with multiple responsibility layers).

4. Define a space with representative structure with respect to common modeling patterns.

Defining a space with representative structure is inspired by the following methods. The method of combinatorial testing in [32] identifies representative combinations of parameters of the system to be tested. The method in [33] proposes testing of model transformations using partial representative models instead of full detailed models. The method in [34] extracts representative behavioral property patterns from an UML/OCL model of the system to generate efficient test cases for testing.

An example of a common modeling pattern is inheritance hierarchies that have up to 4 levels with up to 3 subclasses per class. This is found in a large number of diagrams from the UML superstructure [35] and SysML specifications documentation [36].

The first constructed space assumes that the design is created at logical level, where there are lots of directed associations and few generalization and bidirectional associations. There are no compositions/aggregations. Also, there are very few attributes per class. We pursue a type of system that is hierarchical with respect to dependencies. The space has the following characteristics.

- 10 classes.
- 2 - 3 attributes per class.
- There is at most one generalization.
- There is at most one bidirectional association.
- No compositions/aggregations.
- There is at most one association path between any pair of classes.
- There is at most one direct relationship between any pair of classes.
- The maximum number of relationships per class is 3.
- The maximum association path length between any two indirectly related classes is 3.

Trivial variations of diagram characteristics were minimized as follows.

- For each diagram structure in terms of directed/bidirectional associations and generalizations there are 3 variations of class attributes: 1) all classes have 2 attributes, 2) all classes have 3 attributes, and 3) classes with exactly one relationship have 2 attributes, while all others have 3 attributes.
- For each diagram structure in terms of associations and generalizations the direction of directed associations is fixed and arbitrary.

The second constructed space assumes that the design is created at a logical level, where there are very few attributes per class. We want to consider a system that has lots of abstraction and structure (e.g. UML superstructure). The space has the following characteristics.

- 18 classes.
- 2 - 3 attributes per class.
- There is an inheritance hierarchy and an aggregation hierarchy, each with a

depth of 2 levels and consisting of 7 classes. Aggregations are represented by directed associations.

- There is an association (directed or bidirectional) between two hierarchies.
- There are 4 additional classes connected via directed associations to the inheritance and aggregation hierarchies (2 classes for each hierarchy).
- The maximum number of relationships per class is 3.

Trivial variations of diagram characteristics were minimized as follows.

- The variations of class attributes follow the same pattern as in the first constructed space.
- The association between the two hierarchies is defined over a subset of 3 classes from each hierarchy (instead of all 7 classes). If the association is directed, it always points to the aggregation hierarchy.
- The associations connecting the additional classes to the hierarchies also vary over a subset of 3 classes in each hierarchy. These associations are always directed toward the additional classes.

The experiments were carried out on the Northeastern University Discovery Cluster [37] with node speeds ranging from 1.8 to 2.8 GHz.

For the first constructed space, we generated 250,000 random diagrams and identified 1324 equivalence classes as shown in **Figure 19**. **Figure 20** shows the percentage of class diagrams that were classified where 1) basic templates 1 - 4 are used, 2) basic templates 1 - 3 are used, and 3) basic templates 1 - 3 and a basic template with two directed associations (shown in **Figure 21**) are used. The experimental lower bound of the size of the space (total number of structurally different diagrams) is 71,924. The number of equivalence classes found significantly reduces the number of diagrams to be used from the testing space. For the second constructed space, we generated 160,000 random diagrams and identified 1942 equivalence classes as shown in **Figure 22**. The results of class diagrams that were classified are similar to those in the first space. The experimental lower bound of the size of the space is 5397 while the total size of the space is close to this number. The number of equivalence classes found effectively reduces the number of diagrams to be used from the testing space.

By using basic templates 1 - 4, we were able to classify all the generated class diagrams. This suggests that basic templates 1 - 4 are optimal for the diagram classification method.

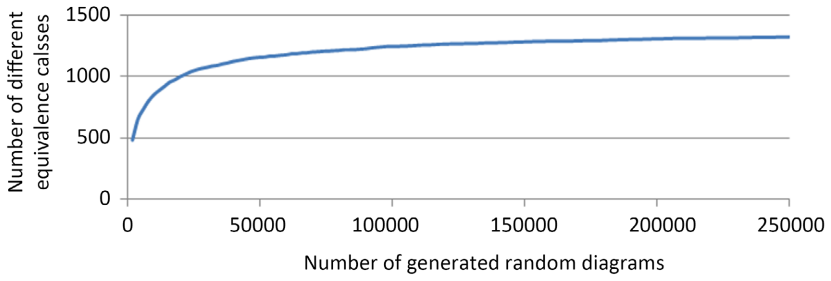


Figure 19. Distribution of equivalence classes of diagrams from the first constructed space.

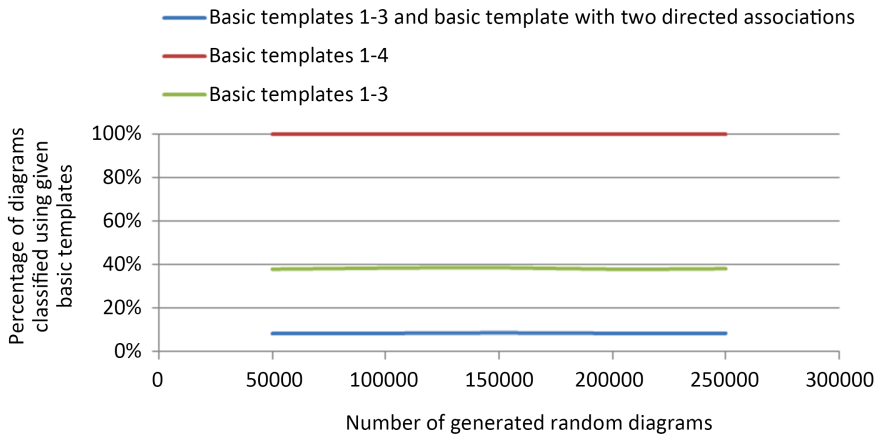


Figure 20. Evaluation of classification of class diagrams from the first constructed space.

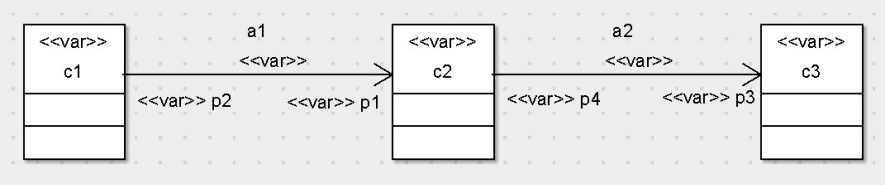


Figure 21. Diagram of basic template with two directed associations.

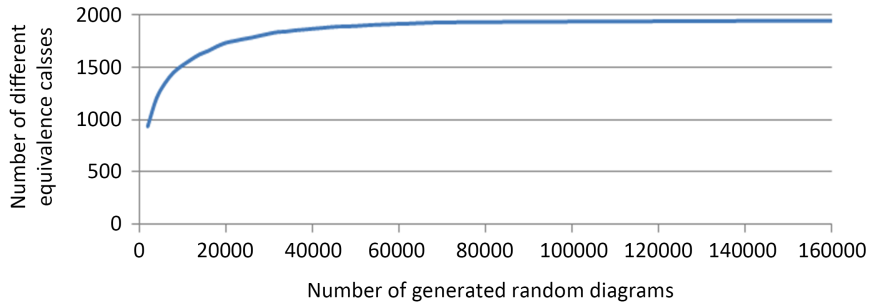
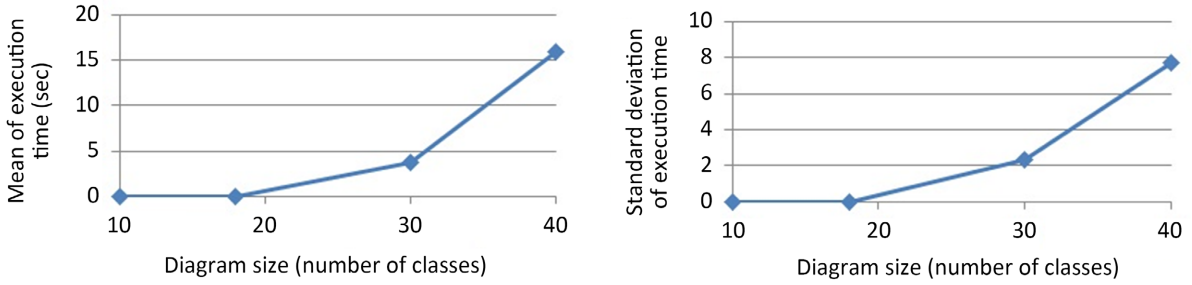


Figure 22. Distribution of equivalence classes of diagrams from the second constructed space.

We also conducted a performance evaluation experiment for diagram classification based on basic patterns 1 - 4. We tested classification of random diagrams with 10, 18, 30, and 40 classes (1000 diagrams for each size). The diagrams with 10 and 18 classes were generated from the first and second constructed spaces. For diagrams with 30 and 40 classes, we constructed two additional spaces similar to the second constructed space. These spaces have more inheritance and aggregation hierarchies. For each set of experiments with random class diagrams of a given size, the arithmetic mean and standard deviation of the execution time were calculated. This is shown in **Figure 23**.

We obtained a reasonable average execution time for the classification of large diagrams with 40 classes. In addition, the standard deviation for these diagrams indicates that, in most cases, the average execution time does not have a significant

deviation. Further investigation of execution times for diagrams of all sizes revealed that it takes less than a second to calculate the *ATGI*-graph isomorphism. The bulk of the execution time comes from the composition of the given basic templates into a complex template. The graph in **Figure 23(a)** suggests a polynomial execution time for this composition. This agrees with the time complexity of the basic template composition algorithm mentioned in Section 6.3.



(a) Arithmetic mean of execution time for random diagrams of given size

(b) Standard deviation of execution time for random diagrams of given size

Figure 23. Performance evaluation of classification of class diagrams of different sizes.

10. Conclusions

We developed a formal method for grouping class diagrams into isomorphic classes with the following properties: 1) the number of isomorphic classes is finite, 2) the isomorphic classes are disjoint, 3) the isomorphic classes are concerned with the structure of class diagrams, 4) the isomorphic classes have a basis (the given collection of basic templates), and 5) the isomorphic classes are based on the UML metamodel. We showed experimentally that any class diagram that is an instance of the minimal UML metamodel and consists of class diagram elements of types covered by the basic templates shown in Section 7 can be grouped into isomorphic classes using our method.

Our method can be used to improve the efficiency of testing operations on class diagrams. A chosen method can be tested using spaces of diagrams that simulate the common structure of real-world diagrams and are efficient for testing in terms of structural diversity and size. The given space is partitioned on isomorphic classes of diagrams. Then at least one diagram from each class can be used for testing.

Another usage of our method is to improve the efficiency of Machine Learning methods that operate on class diagrams. The training of a model can be done on spaces of diagrams that simulate the common structure of real-world diagrams and are efficient for training with respect to structural diversity and size. The given space is partitioned on isomorphic classes of diagrams. Then one diagram from each class can be used for training.

Our method is beneficial for finding groups of similar refactored diagrams in a large set of diagrams. This is important for design version control. It also supports managing multiple concretized versions of a design [38], because it can detect similar detailed diagrams that refine the same abstract idea.

Using our method, it is possible to evaluate the space of class diagrams for the structural diversity of diagrams. A diverse space should include a large number of classes of class diagrams.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Awodey, A. (2010) Category Theory. Oxford University Press.

[2] Leśniewski, S. (1992) A General Theory of Manifolds, Collected Works. Kluwer.

[3] Hovda, P. (2008) What Is Classical Mereology? *Journal of Philosophical Logic*, **38**, 55-82. https://doi.org/10.1007/s10992-008-9092-4

[4] Tazin, A. and Kokar, M.M. (2022) Composition of UML Class Diagrams Using Category Theory and External Constraints. *Journal of Software Engineering and Applications*, **15**, 436-468. https://doi.org/10.4236/jsea.2022.1512025

[5] Tazin, A. (2022) Composition of UML Class Diagrams Using Category Theory and External Constraints. Ph.D. Dissertation, Northeastern University.

[6] Mangaroliya, K. and Patel, H. (2020) Classification of Reverse-Engineered Class Diagram and Forward-Engineered Class Diagram Using Machine Learning.

[7] Maneerat, N. and Muenchaisri, P. (2011) Bad-Smell Prediction from Software Design Model Using Machine Learning Techniques. 2011 8*th International Joint Conference on Computer Science and Software Engineering* (*JCSSE*), Nakhonpathom, 11-13 May 2011, 331-336. https://doi.org/10.1109/jcsse.2011.5930143

[8] Halim, A. (2013) Predict Fault-Prone Classes Using the Complexity of UML Class Diagram. 2013 *International Conference on Computer*, *Control*, *Informatics and Its Applications* (*IC3INA*), Jakarta, 19-21 November 2013, 289-294. https://doi.org/10.1109/ic3ina.2013.6819188

[9] Al-Khiaty, M.A. and Ahmed, M. (2016) UML Class Diagrams: Similarity Aspects and Matching. *Lecture Notes on Software Engineering*, **4**, 41-47. https://doi.org/10.7763/lnse.2016.v4.221

[10] Pedersen, T., Patwardhan, S. and Michelizzi, J. (2004) WordNet: Similarity: Measuring the Relatedness of Concepts. *HLT-NAACL—Demonstrations* '04: *Demonstration Papers at HLT-NAACL*, Boston, 2-7 May 2004, 38-41. https://doi.org/10.3115/1614025.1614037

[11] Chechik, M., Nejati, S. and Sabetzadeh, M. (2011) A Relationship-Based Approach to Model Integration. *Innovations in Systems and Software Engineering*, **8**, 3-18. https://doi.org/10.1007/s11334-011-0155-2

[12] Costa, V. (2014) Detecting Semantic Equivalence in UML Class Diagrams, SEKE.

[13] Elasri, H., Elabbassi, E., Abderrahim, S. and Fahad, M. (2018) Semantic Integration of UML Class Diagram with Semantic Validation on Segments of Mappings.

[14] Rao, R.S. and Gupta, M. (2013) Design Pattern Detection by Greedy Algorithm Using Inexact Graph Matching. *International Journal of Engineering Research and Technology*, **2**, 3658-3664.

[15] Rao, R.S. and Gupta, M. (2013) Design Pattern Detection by Sub Graph Isomorphism Technique. *International Journal of Engineering and Computer Science*, **2**, 3101-3105.

[16] Pradhan, P., Dwivedi, A.K. and Rath, S.K. (2015). Detection of Design Pattern Using Graph Isomorphism and Normalized Cross Correlation. 2015 8*th International Conference on Contemporary Computing* (*IC*3), Noida, 20-22 August 2015, 208-213. https://doi.org/10.1109/ic3.2015.7346680

[17] Chartrand, G. (1985) Isomorphic Graphs. Dover Books.

[18] Rao, R.S. and Gupta, M. (2013) Design Pattern Detection by a Heuristic Graph Comparison Algorithm. *International Journal of Advanced Research in Computer Science and Software Engineering*, **3**, 251-255.

[19] de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U. and Taentzer, G. (2007) Attributed Graph Transformation with Node Type Inheritance. *Theoretical Computer Science*, **376**, 139-163. https://doi.org/10.1016/j.tcs.2007.02.001

[20] Golas, U., Lambers, L., Ehrig, H. and Orejas, F. (2012) Attributed Graph Transformation with Inheritance: Efficient Conflict Detection and Local Confluence Analysis Using Abstract Critical Pairs. *Theoretical Computer Science*, **424**, 46-68. https://doi.org/10.1016/j.tcs.2012.01.032

[21] Tazin, A. (2024) Class Diagram Classifier. https://github.com/alextazin/DiagramClassifier

[22] Michail, D., Kinable, J., Naveh, B. and Sichi, J.V. (2020) JGraphT—A Java Library for Graph Data Structures and Algorithms. *ACM Transactions on Mathematical Software*, **46**, 1-29. https://doi.org/10.1145/3381449

[23] Cordella, L.P., Foggia, P., Sansone, C. and Vento, M. (2004) A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **26**, 1367-1372. https://doi.org/10.1109/tpami.2004.75

[24] Elseidy, M., Abdelhamid, E., Skiadopoulos, S. and Kalnis, P. (2014) GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph. *Proceedings of the VLDB Endowment*, **7**, 517-528. https://doi.org/10.14778/2732286.2732289

[25] NetworkX Developers (2024) NetworkX. https://networkx.org/

[26] Peixoto, T.P. (2025) The Graph-Tool Python Library. https://graph-tool.skewed.de/

[27] Helsinki University of Technology (2021) Bliss: A Tool for Computing Automorphism Groups and Canonical Labelings of Graphs. http://www.tcs.hut.fi/Software/bliss/

[28] Junttila, T. and Kaski, P. (2007) Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. In: 2007 *Proceedings of the* 9*th Workshop on Algorithm Engineering and Experiments* (*ALENEX*), Society for Industrial and Applied Mathematics, 135-149. https://doi.org/10.1137/1.9781611972870.13

[29] Junttila, T. and Kaski, P. (2011) Conflict Propagation and Component Recursion for Canonical Labeling. In: Marchetti-Spaccamela, A. and Segal, M., Eds., *Theory and Practice of Algorithms in* (*Computer*) *Systems*, Springer, 151-162. https://doi.org/10.1007/978-3-642-19754-3_16

[30] Evans, E. (2003) Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional.

[31] Bennett, S., McRobb, S. and Farmer, R. (2010) Object Oriented Systems Analysis and Design Using UML. McGraw-Hill Higher Education.

[32] Cohen, D.M., Dalal, S.R., Fredman, M.L. and Patton, G.C. (1997) The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering*, **23**, 437-444. https://doi.org/10.1109/32.605761

[33] Sen, S., Mottu, J., Tisi, M. and Cabot, J. (2012) Using Models of Partial Knowledge to

Test Model Transformations. In: Hu, Z.J. and Lara, J., Eds., *Theory and Practice of Model Transformations*, Springer, 24-39. https://doi.org/10.1007/978-3-642-30476-7_2

[34] Dadeau, F., Fourneret, E. and Bouchelaghem, A. (2017) Temporal Property Patterns for Model-Based Testing from UML/OCL. *Software & Systems Modeling*, **18**, 865-888. https://doi.org/10.1007/s10270-017-0635-4

[35] OMG (2011) UML Superstructure. http://www.omg.org/spec/UML/2.4.1/

[36] OMG (2024) SysML. https://www.omg.org/spec/SysML/2.0/Beta2/About-SysML

[37] Northeastern University (2024) Discovery Cluster. https://rc.northeastern.edu/compute/

[38] Tazin, A., Lu, S., Chen, Y., Kokar, M.M. and Smith, J. (2022) Modeling Concretizations in Software Design. In: Lee, R., Ed., *Software Engineering Research, Management and Applications*, Springer International Publishing, 47-65. https://doi.org/10.1007/978-3-031-09145-2_4