Towards Verifying UML Class Diagram and Formalizing Generalization/Specialization Relationship with Mathematical Set Theory

Kruti P. Shah*, Emanuel S. Grant
School of Electrical Engineering and Computer Science, University of North Dakota, Grand Forks, ND 58201, USA.

*Corresponding author. Email: kruti.shah@ndus.edu; emanuel.grant@ndus.edu
Manuscript submitted July 23, 2022; Accepted November 18, 2022.
doi: 10.17706/jsw.17.292-303

Abstract: The Unified Modeling Language (UML) is considered the de facto standard for object-oriented software model development. This makes it appropriate to be used in academia courses at both the graduate and undergraduate levels of education. Some challenges to using the UML is academia are its large number of model concepts and the imprecise semantic of some of these concepts. These challenges are daunting for students who are being introduced to the UML. One approach that can be taken in teaching UML towards addressing these concerns is to limit the number of UML concepts taught and recognize that students may not be able to develop correct UML system models.

This approach leads to research work that develop a limited set of UML model concepts that are fewer in number and have more precise semantics. In this paper, we present a new approach to resolve an aspect of this problem by simplifying the generalization/specialization semantics of the class diagram through the application of mathematical formality to usage of these class diagram concepts. Along with that, we discuss the progress of research in the area of verification of UML class models. This research work derives a core set of concepts suitable for graduate and undergraduate comprehension of UML modeling and defines more precise semantics for those modeling concepts. The applicable mathematical principles applied in this work are from the domains of set theory and predicate logic. This approach is particularly relevant for the pedagogy of software engineering and the development of software systems that require a high level of reliability.

Key words: Formal specification techniques, mathematical set theory, UML class diagram.

1. Introduction

The UML (Unified Modeling Language) [1] is a graphical modeling language used to specify, simulate, and construct software system components. The UML has been adopted and standardized by the Object Modeling Group [2]. UML is considered the standard for object-oriented software model development that allows modeling various aspects of complex systems [2]. However, there are concepts in the UML with imprecise semantics, which makes it challenging to using it in graduate and undergraduate courses on software development. In addition to the challenges to teaching with the UML, developing technologies for analysis and verification of UML models is significant to developers who use UML for system modeling.

This work considers the UML class diagram, which is the most fundamental and widely used l UML model.

A Class Diagram provides a static representation of system components. The purpose of a class diagram is to display classes with their attributes and methods, hierarchy (generalization) class relationships, and associations (general, aggregation, and composition) between classes in one model [3].

On any student software development projects, team members will have to communicate between themselves and the instructor of the course. In some instances, communication will be conducted with system models, such as the UML class diagram. In model-driven development, models are refined and transformed through the various life cycle activities. With an imprecise semantics for model component, the process may result in an incorrect mapping between model components and code components. In a similar manner, industrial software development projects are hampered by incorrectly designed models and faulty code developed from those models.

An approach to resolving this problem is to make precise the semantics of the class diagram concepts through the application of mathematical formality to the definition and usage of these class diagram concepts. The applicable mathematical principles result in a reduction of complexity in the UML class diagram and leads to a better understanding of the model. A precursor to the application of formalism to the model concepts is the elimination of redundant components in UML class diagram. The mathematical principles applied in this work are from the domains of set theory and predicate logic, which are particularly relevant for the development of software systems that require a high level of reliability. This work will also be beneficial to software teaching and industrial development, as a simpler and more precise set of software modeling components should lead to greater appreciation of modeling strategies.

The main goal of this research is to derive means to verify the correctness of generated UML class diagram models. Formal specification methods are feasible solution to achieve this goal. In order to accomplish that as a first step, this research attempts to reduce the number of concepts in developing class diagram models. The reduction of the number of model concept has a two-fold purpose. Firstly, the elimination of model concepts can be accomplished by expressing a model concept in terms of other model concepts. Secondly, reducing the number of model concepts reduces the number of possible model concept combinations; this leads to simplified modeling constructions.

The focus at this first stage of the research is to eliminate the generalization relationship by applying mathematical principles then representing the generalization/specialization UML model concept in terms of the general association concept, with multiplicity constraints. This work reported in this paper identifies the first step of this research. The outcome from this research will be applied in graduate and undergraduate software engineering courses, and demonstrated on assessing the correctness of models used in error handling in the field of automatic code generation (program synthesis systems) [4], with specificity to various safety-critical systems.

The remainder of the paper is organized as follows. Section 2 gives a brief theoretical background of UML models and describes set theory and relationships, which can be used to remove duplicity associated with UML models. Section 3 includes the overview of related work in the area of verification and correctness of UML models. Section 4 demonstrates the methodology used for simplifying the semantics of the class diagram concepts. The paper concludes with a look at future work.

2. Background

This section covers some of the theories and prior work in the area of UML model transformation and mathematical representation of such models.

2.1. Unified Modeling Language (UML)

The Unified Modeling Language (UML) [2] has been widely accepted as the standard language for modeling and documenting software systems. The UML significance has been enhanced with the beginning of the

Model-Driven Development (MDD) approach, in which analysis and design models play an essential role in the process of software development. The UML offers a number of models used to describe particular aspects of software artifacts. These diagram structures can be divided into two categories static and dynamic views:

Static view: It describes the structural aspect of the system and its components. It includes objects, classes, attributes, operations, and their inter-relationships. The structural view can be represented by class diagrams and composite structure diagrams.

Dynamic view: It describes the behavioral aspect of the system. Dynamic view reflects the changes related to the internal states of individual objects and changes in the overall state of the system. This view can be represented as sequence diagrams, activity diagrams, and state chart diagrams.

2.2. UML Class Diagram

The UML class diagrams are used to represent the static structure of the system components [2]. It describes the structure of system in terms of classes, attributes, and constraints imposed on classes (operations) and their inter-relationships. This work focuses on use of the UML class diagrams. Class diagrams are used at the analysis phase to present a view of the static entities in the problem domain, and at the design phase to present a view of the static entities (classifiers) in the solution domain. A class diagram is best described as a set of graph elements connected by their relationships. Classes in UML models are represented as rectangles. Each class consists of a name, a set of attributes, and a set of operations on the class's attributes. Figure 1 shows an example of a class diagram consisting of classes, associations (aggregations and compositions), and generalizations.

2.2.1. UML associations (aggregation, association, composition, generalization)

There are rules and requirements for combining the classes to construct partial or complete UML class models. Association, which may be depicted as general association, is illustrated as a relationship between Class A and Class C of Figure 1. The association lines indicate the possible relationship between the class entities [5]. An association represents the relationships that involve attributes and methods from the related classes, such as the relationship between class A and class C seen in Figure. 1. Association ends can be annotated with labels, known as Association End names and multiplicities. Multiplicity can be expressed as specific numbers or range of numbers, as shown in Figure 1 between classes A and C.

Aggregation: An aggregation is represented as an association with a white diamond on one end, where the class at the diamond end is the aggregate (container class) and it includes or owns instances of the class (contained class) at the other end of the association [5] (e.g., relationship between class A and B in fig. 1).

Composition: It is a special type of aggregation, in which instances of the contained class are explicitly owned by instances of the container classes [5]; if an instance of the container class is deleted, the instances of the contained class are also deleted. Figure 1 illustrates class C, the container class, and class D, the contained class. It represents composite association as a black diamond.

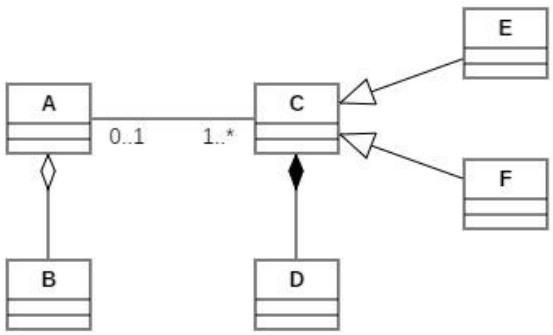

Fig. 1. Sample UML class model.

Generalization: A generalization is represented by an association with a triangle on one end, where the class at the triangle end of the association is the parent class and at the other ends of the association represents the subclasses [5]. A subclass inherits all of the parent class's attributes, operations, and associations (e.g., subclasses E and F inherit properties of parent class C in Figure 1).

2.3. Model Transformation

Models provide a level of abstraction that allows developers and stakeholders to visualize different parts of the system while avoiding implementation details. A large number of models can exist for any given system, and it is essential to assure the consistency of those models [6].

Most software engineering operations have included model transformation in their development life cycle. It is the process of transforming a graphical model for the purposes of analysis, optimization, evolution, migration, or even code generation. Model transformation employs a collection of rules known as transformation rules, which take one or more input models and output one or more target models [7].

Model transformation can be either manual or automatic. Manual transformation involves an application of custom transformation rules while in automatic transformation the predefined transformation rules are applied to class model [7]. Regardless of the transformation method used, it is essential that the software engineer has a thorough understanding of the project's scope, as well as the syntax and semantics of the source and target models. Transformation rules will be designed and applied to the models in order to automate the transformation process. The source models will be UML class diagrams, and the target models will be their equivalent formal specification schemas.

2.4. Formal Specification Methods

The inadequacies of system and software specifications are one of the primary issues with software-intensive systems. Although the requirements should usually accurately describe the functions of the software system, many of the details that should be carried out and defined in a more detailed specification are not addressed.

As a result, there are inconsistencies and misinterpretations, which lead to issues in the latter stages of design and implementation. These issues are frequently identified during the system integration stages. There are graphical software development methods, such as data-flow diagrams, finite state machines, and entity relationship diagrams, that have been shown to help with the development of better specifications, but they lack precision in the details of the specification and a smooth way of developing a design and implementation.

Formal specification methods are feasible solution to these issues. They precisely define the system and ensure a smooth transition from specification to design to implementation. There are a number of formal specification languages such as Z notation, Object Constraint Language (OCL), VDM, Alloy etc. In general, all of these formal specification languages involve formal specification, refinement, and verification, which comprise of set theory, predicate logics, and algebra, among other things.

The syntax and semantics of static and dynamic aspects of a system are formally specified in terms of mathematical notations in formal languages. Formal languages improve the system's reliability and security by reducing ambiguity in the system's requirements using their mathematical representation. The use of formal languages is essential while working with the large/complex real-time software systems in which the accuracy of the system is important.

2.5. Set Theory and Relationships

A goal of this work is to simplify and formalize the concepts used in the development of class diagram models. In order to do that, model concepts that may be represented by other model concepts are eliminate

by establishing mathematical equivalency between the two model concepts. Once such concept, target in this work is the UML relationship of generalization/ specialization (parent/children).

Table 1 presents the name of the set theory function, as well as the corresponding multiplicity and a description of the relationship. The arrows ↣, ⇸, ⇻, and ⤔ with barbed tails form injective sets of functions [8]. The arrows ↠, ⇸, and ⇻ with double heads form surjective sets of functions [8]. The functions X↣Y, X⇸Y, X⇻Y, and X⤔Y map the elements of X onto the elements of Y in a one-to-one relationship. Each function has associated constraints, which map elements of their domain to elements of their range.

Table 1. Relational Definitions [6]

Function		Constraints		
Name	Symbol	dom f	*OneOne*	ran f
Total function	$\rightarrow$	$= X$		$\subseteq Y$
Partial function	$\nrightarrow$	$\subseteq X$		$\subseteq Y$
Total injection	$\rightarrowtail$	$= X$	*Yes*	$\subseteq Y$
Partial injection	$\nrightarrow$	$\subseteq X$	*Yes*	$\subseteq Y$
Total surjection	$\twoheadrightarrow$	$= X$		$= Y$
Partial surjection	$\nrightarrow$	$\subseteq X$		$= Y$
Bijection	$\rightarrowtail$	$= X$	*Yes*	$= Y$
Partial bijection	no symbol defined	$\subseteq X$	*Yes*	$= Y$
Finite partial function	$\nrightarrow$	$\subseteq X$		$\subseteq Y$
Finite partial injection	$\nrightarrow$	$\subseteq X$	*Yes*	$\subseteq Y$

3. Related Works

In this section, several studies related to the verification and correctness of UML models are described and related to the work presented in this report.

Taentzer proposed a graph transformation tool (AGG) [10] that defines rule-based manipulation of graphs. It has visual editors for graphs, graph grammars as well as the formal foundation based on the algebraic approach to graph transformation. AGG also offers support for model validation techniques for graph grammars. It consequently implements the theoretical results available for algebraic graph transformation to support their validation. However, it does not support all UML features (e.g., association classes). The specification techniques found in graph grammars and transformation languages [10] were not sufficient for the purposes of the research reported on in this paper, as they do not fully comply with UML concepts.

Zhao *et al*. [11] described a general transformation approach of UML diagrams into Petri nets based on meta-modelling and graph transformation techniques. They formally transform UML state charts and behavioral diagrams to Petri nets for verification. They identified three layers of relationship among various UML diagrams: the relationships among the same UML diagram from different contextual instances; the relationships among various diagrams from the same view of a system; and the relationships among various diagrams from different views of a system. However, a drawback still persists in the modeling of large problems as in this work the authors only considered experiments on the verification of simple UML state chart diagrams. Also, the third layer which describes the relationship between the diagrams of static structure view and the diagrams of dynamic behavior view is rarely considered in the available works related to verification and transformation of UML models.

Chama *et al*. [12] have proposed an automatic approach and a tool to check UML models using graph transformations. They considered both static and dynamic models for inconsistency checking. The idea was to map class diagrams, state charts and communication diagrams into an equivalent Maude specification. It uses a meta-modelling approach that could help in model checking. The authors used a subset of UML diagrams to develop an AToM3 integrated framework for their model checking by transforming them into a rewriting system expressed in the Maude language and graph grammars. The formal verification is

performed using the Linear Temporal Logic (LTL) Model Checker. However, the UML models used for Maude language and LTL model verification were incorrectly drawn. Since UML models are ambiguous, validated models that are incorrectly specified may be ambiguous as well.

In many software engineering courses, drawing UML diagrams, such as class diagrams, is an essential piece of work. Students are required to draw models that depict system requirements as part of their coursework. These diagrams are usually graded manually by the course instructor. In UML, there could be several valid solutions for the same system requirements. Therefore, it requires a great amount of time for grading. As a result, an automated grading tool that assists the instructor in the grading process is required. Although, there is a significant amount of work done in the area of automatically grading UML models [13-16].

Outair *et al*. [13] proposed a student diagram assessment system that provides a verification mechanism wherein the teacher manually compares his/her solution with the ones designed by the student. At the end of the comparison process, the system generates a list with the differences and comments that can be used by the student to improve his/her diagram. The contribution revealed in this work, is the proposal of a transformation method of class diagram into a graph using UML metamodel.

In [14], Herout et al. introduced UMLTest a java-based UML class diagram test tool with the aim of assessing student diagrams developed for object-oriented courses. For this approach, teacher constructs their UML class model using the UMLet tool, which is assumed to be an error-free solution. The UMLTest tool takes XML files as input and generates test solutions as output.

Salisu *et al*. [15] have proposed an automated java-based student UML evaluation tool that assesses students' UML class models by comparing them with the instructor's solution. The tool accepts two zip files as input, one for a student and another one for the instructor in XMI format. In this approach, instructors and students are required to design a class model using the Modelio UML tool. The tool creates two feedback files: one for the students, which contains student marks and some guideline information about the assessed diagram, and another one for the tutor, which contains all of the student information as well as their final mark for each diagram.

Bian et al. [16] have developed an automatic class diagram grader tool TouchCore using Eclipse modelling framework that can be used to compare a student solution to an instructor solution and assign grades with the basic comments (e.g., missing class) based on a customizable grading scheme. This method uses syntactic, semantic, and structural matching to create mappings between instructor and student solutions and provides a final grade with valuable comments. However, there are certain limitations associated with the proposed work [13]-[16]. Neither of these approaches contains a case study to assess the UML class model including aggregation, composition, or generalization/specialization relationship. Therefore, it is unsure how efficiently it will handle a generalization/specialization relationship or complex UML class models. In addition, the teacher's UML class model is designed manually using tools such as Modelio, UMLet, StarUML, etc. As a result, there might be a chance of ambiguities, uncertainties, or errors in the teacher's models. As a result, a tool should first support the validation of teacher's models using formal methods [17] in order to verify the correctness of the student's models.

The Z notation is used [18]–[21] to formalize and verify the UML class model. The authors (Evans et al.) employed Z notation to develop the formal foundation for the UML core metamodel. They claimed that the formal foundation provides a number of benefits, including transparency, extendability, consistency testing, refinement, and proof [18], [19].

Evans et al. have defined a compositional schema with multiple subschemas to represent the UML class model. The sub-schemas formalize UML model elements such as type, instance, values, operation, associations, generalization etc. The authors also propose three alternatives for formalizing the UML model [18]: 1) Supplementary: In this way, the UML model's informally specified elements are formally expressed.

2) Object-Oriented Extended Formal Language: In this approach, established formal methods are extended with object-oriented principles such as Object-Z and Z++. 3) Method Integration: In this method, the complete UML model is translated into a formal model in order to improve its precision.

The authors of [18] expanded on their previous work by proposing a graphical representation transformation of the UML class model. They also offered a three-step roadmap for formalizing and verifying models: 1) Select a formal language that is both expressive and well supported by the tools for the model's static and dynamic features of UML class model. 2) Formally describe a graphical modeling notation's abstract syntax. 3) Define a function that transforms the model's syntax and semantics into formal notation. Finally, tools for validating formal semantics should be developed.

The authors of [20] suggested that formal UML analysis alone is insufficient for determining semantic correctness. Furthermore, the authors stated that it is not particularly accessible to practitioners with limited knowledge of discrete mathematics, and that industry experts' comments is also necessary for the semantic validity of the UML model. In [21], Authors designed a formal methods reference manual for Z notation, which precisely and explicitly specifies the semantics of UML concepts. Along with that, the Inference rules for examining various UML model properties are provided in the reference manual [21].

In [22] – [28], object constraints language (OCL) used for verification of the UML class model. Cadoli et al. [16] proposed a constraint programming-based linear inequality-based method for finite model verification. They used the Constraint Satisfaction Problem (CSP) to represent the UML class model, and the ILOG's Solver assessed the satisfiability of the UML class model [22]. The Managed Object Format (MOF) syntax is used by the ILOG solver as an input. In addition, two class model correctness issues were addressed and encoded into CSP. In the first problem, they check that all the model's classes are completely satisfied at the same time. In the second problem, they prove that a finite non-empty model can be generated from the class model.

To verify the UML class model, Malgouyres and Motet [23] employed Constraint Logic Programming (CLP). They used CLP clauses to translate the UML class model, metamodel, and meta-metamodel [23]. In this approach, concrete metamodel and UML class model elements are translated into CLP facts while abstract elements and constraints are transformed into rules. CLP's goals are also specified, which contradicts the consistency standards. Finally, the inconsistencies are handled by a unified checker. The UML class model is considered inconsistent if the unified checker identifies the solution to the goal and if the goals are resolved.

Pérez and Porres [24] proposed a system for using CLP to assess the satisfiability of a UML class model. The suggested methodology detects design flaws in UML class models with OCL annotations. They used the bounded verification approach and used the model-finding tool formula to reason about finite constraints for the number of instances of the model. The suggested method verifies predefined correctness features such as satisfiability and the lack of redundant constraints. It can also be used to analyze complex models in order to discover the optimal object model for the domain. They also used an eclipse plug-in called CD-to-Formula to design the proposed framework.

Cabot et al. [25] presented incremental verification of the class model's OCL integrity constraint. Integrity checking is a technique used for determining whether an operation violates a specified integrity constraint. They introduced the term Potential Structure Event (PSE) and stated that verifying integrity requirements after each structure event (e.g., Insert, Update, Delete, or Specialized Entity) can be costly and time-consuming [25]. As a result, PSEs for each integrity constraint are recorded, and only those events that can violate the constraint are represented. Furthermore, only the instances of entity and relationship types that have been affected by PSEs are validated and verified.

Cabot *et al*. [26] presented an approach to translate UML class models annotated with OCL constraints into a constraint satisfaction problem (CSP). The authors briefly discussed translation of UML/OCL classes, associations, generalization sets, and OCL invariants into CSP. A tool based on CSP [27] is then used to verify

a predefined set of correctness properties for the original UML/OCL diagrams. The UML/OCL language combination integrates well with automated inference systems. If the generated CSP is solvable, the model is considered satisfiable otherwise is considered unsatisfiable. The CSP tool supports bounded reasoning about satisfiability, consistency, finite satisfiability, independence of invariants, and partial state completion. It handles class diagrams with multiplicity, class hierarchy, association-class constraints but does not allow multiple inheritance. Along with that, tool does not support all the features in OCL specification, such as constraints on a string, aggregation, and composition relationship.

Cabot *et al.* presented the UMLtoCSP tool in [27]. It takes the XMI format for the class model and OCL as a separate text file for input. The model and OCL are translated to CSP, which is then verified by the CSP solver. The XMI file is parsed using the Metadata Repository API, while OCL constraints are processed by the Dresden OCL Toolkit.

Cabot *et al.* [28] expanded on their previous work [26], arguing that an insufficient constraint or bound could miss defects in the model due to a small search space or could be inefficient if set too large. Large initial bounds and constraints are set in the proposed solution [28]. Then, using the interval constraint propagation technique, the set of bounds is tightened up as much as feasible with user input, and unwanted value from the bounds is removed. Since then this technique has been enhanced to the point where verification bounds are now defined automatically whenever its possible.

Both the methods Z and OCL [18] – [28] provide support for association, aggregation, and generalization relationships but do not support the features like dependency relationships (aggregation and composition) and xor constraint. Z notation is supported by Z word and Z/Eves verification tools. USE and UMLtoCSP tools are capable of working with OCL. Both of these tools support semi-automatic transformation. Both the tools (Z word and USE) provide feedback to users in order to notify them of the verification process's outcome. However, this models or tools require from the user a significant level of expertise on formal aspects in order to understand the feedbacks and resolve the issues. Finally, efficiency is a major concern. Current UML class model verification methods effectively verify the correctness of small models with few constraints. However, in some circumstances, especially when dealing with large and complex models, their performance suffers. Along with that, they also lack support for certain key features of the UML class model.

In our opinion, a verification tool, in order to be effective and widely adopted, has to present, at least, few important characteristics: 1) It should provide support for some key features of UML class model (i.e., aggregation, composition, xor constraint), 2) It should easily integrate into the model designer tool chain, 3) It should offer meaningful feedback for the user, and 4) It should be relatively efficient while verifying the large or complex real-world UML class models.

4. Methodology

This work attempts to reduce the number of concepts used to develop UML class diagram models. An approach to resolve this problem is to simplify the semantics of the class diagram concepts through the application of mathematical formality to the definition and usage of class diagram concepts. The main goal here is to remove duplicity associated with the generalization relationship by applying mathematical principles and set theory. This work of reducing the number of model concepts is a precursor to the second stage of the research of transforming the UML class diagram to a graph representation in order to formally analyze the structure of the model. A second benefit of this first phase of the research effort, i.e. the reduction of the number of model concepts, is that of working with a smaller number of model concepts will reduce the possible number of errors in developing the models. The number of possible combinations from using ‘n’ number of elements is reduced from 2n - 1 to 2(n-1) - 1 with the elimination of one element.

In this approach, set theory is used to demonstrate a mathematical equivalence of generalization and

general association. Also, relational description of generalization and specialization relationship are provided.

4.1. Generalization and Specialization in UML Class Diagram

In the object-oriented paradigm, generalization/ specialization relationships are used to connect two or more model elements of the same type. It specifies a relationship between a general element and more specialized ones to enhance reusability of model classes. The concept of generalization/specialization relationship is well understood when it is applied to sets. However, several contradictions and ambiguities arise when these concepts (i.e., generalization/specialization) are applied to software model elements.

A class diagram with a generalization/specialization relationship is shown in Figure 2. The class A generalizes the classes B and C. Conversely, classes B and C are specializations or subclasses of class A. Here, classes B and C inherit the properties of class A.

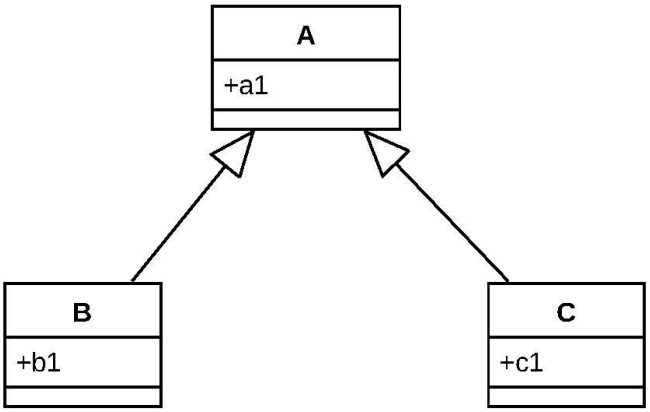

Fig. 2. Generalization and specialization in UML diagram.

To reduce the duplicity associated with UML class diagram concepts and to represent generalization/specialization relationship as a general association, a relationship between the parent (generalization) class and its subclasses (specialization classes) is defined, in which members of class B (i.e., subclass) map bijection relationship (GAB) with elements of class A (i.e., generalize class). This implies, if attributes (a1, b1) of classes A and B belong to GAB then a1 must be b1.

The bijection relationship between classes A and B is shown in the equation below:

$GAB = \{\forall\ (a1, b1) \in A \rightarrowtail\!\!\!\!\!\twoheadrightarrow B \Rightarrow a1 \equiv b1\}$

Similarly, members of class C (i.e., subclass) map bijection relationships (GAC) with elements of class A (i.e., generalize class). This implies that if all attributes (a1, c1) of the classes A and C belong to GAC then a1 must be c1. Bijection relationship GAB and GAC are both one-to-one and onto functions. The bijection relationship between classes A and C is shown in the equation below:

$GAC = \{\forall\ (a1, c1) \in A \rightarrowtail\!\!\!\!\!\twoheadrightarrow C \Rightarrow a1 \equiv c1\}$

A side effects of having multiple concepts in UML models is complexity and by reducing the duplicity associated with UML model concepts, one of the objectives is accomplish, i.e., coping with complexity related to conducting software engineering modeling with many-to-many relationships. Here, an approach is propose for mapping generalization as general association. In this work, the Abstract Class concept is considered another UML class diagram concept that can be avoided in software development efforts. Abstract classes specify that the objects of the subclasses are the only objects of the generalized class in the system. This can be represented by placing a constraint note on the subclasses.

4.2. Demonstrating Mathematical Equivalence with Set Theory

Set theory is the mathematical theory of well-defined collections of sets and objects that are called members or elements of the set. In this section, set theory will be used to demonstrate the relationship between generalization/specialization and mathematical sets, and prove their equivalence using Venn diagram [29]. The purpose of the Venn diagram is to illustrate the mathematical relationship between classes in the UML generalization/specialization relationship.

In the sets A = {1, 2, 3, 4, 5}, B = {1, 2, 3}, and C = {4, 5}, sets B and C are subsets of set A since all their elements are also elements of set A, as depicted in Figure 3. Sets B and C are disjoint sets, as they do not share any common elements. The relational Venn diagram induced by Figure 3 is shown in Figure 4. It maps the relationship between sets A, B, and C. The set A forms a bijection relationship with sets B and C. All elements of set A can be put in a one-to-one correspondence with the elements of sets B and C. The relationship between sets A and B can be written as, {a1 ↔ b1, a2 ↔ b2, a3 ↔ b3}. Similarly, relationship between sets A and C can be written as {a4 ↔ c4, a5 ↔ c5}.

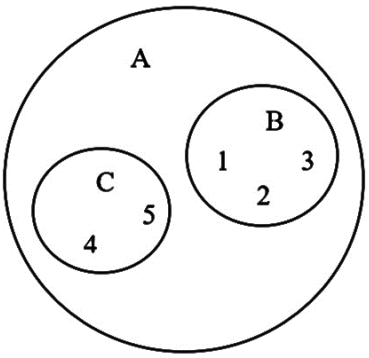

Fig. 3. Venn Diagram of Sets A, B, and C

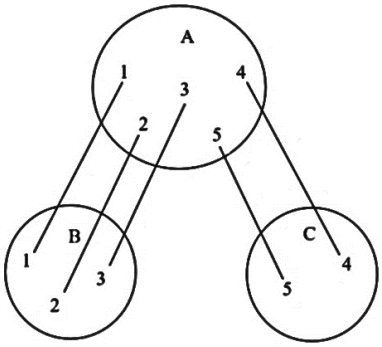

Fig. 4. Relational Venn Diagram

The relationships demonstrated for generalization/specialization as a general association are equivalent to the relationships of sets represented by the Venn diagram. From this demonstration of equivalency between the use of regular UML generalization/specialization relationship and the approach of regular UML association, with bijective constraint, it is determined that only one approach is needed. This generates alternative views of UML models and accomplishes the goal of reducing the number of UML class diagram model concepts, without losing the expressiveness of the UML language. The benefit of this work is fully realized when developers seek to conduct model checking of UML class diagrams and further formal specification analysis [30] in developing safety-critical systems. With a reduced number of model concepts, the process of model checking, and formal specification analysis can be conducted with reduced complexity.

An additional benefit of the use of this alternative approach to regular UML generalization/specialization relationship is realized in teaching introductory software engineering courses, as the reduction in the number of unique concepts, leads to a reduction in the learning curve for the students.

5. Discussion

Software engineering had its official beginning at a NATO Science Committee conference in the 1960s. Since then there have been a number of software development methodologies and standards, which initially were useful but led to complexities in cross-project usage. The issues with the multiple methodologies, notations, and standards led to a coalescing of the major ones into a single graphical notation for software system modeling, i.e. the UML, Unified Modeling Language [2]. This coalescing into a single modeling notation from multiple ones had the side effect of producing a set of related model concepts that are not precise in their semantics. The practice of software engineering accommodates this imprecise semantics in multiple ways, one being formal specification techniques [17]. With the expanding use of software systems in new fields, specifically in the safety critical domain, research needs to advance software engineering towards being an engineering discipline, at the industrial and academic levels.

6. Conclusion and Future Work

UML is the standard for object-oriented software model development that allows modeling and simulating different aspects of software system components. There are many concepts in the UML models with imprecise semantics, which enhance the duplicity and reduce the quality of the UML models. It is important to verify the correctness of UML models. A goal of this work is to reduce the number of concepts in developing class diagram models, towards simplifying model design. Discussion of theoretical background of UML models was conduction and an approach to address imprecision in the UML class diagram model concepts with set theory and their relationships was outlined. This approach is used to eliminate the generalization/specialization relationships and duplicity associated with UML class diagram models. A relational description of the generalization/specialization relationship to demonstrate the mathematical equivalence of generalization and general association that was integrated to determine significant higher-level abstractions for the recovery of UML models was presented.

Future work will focus on applying mathematical theorems and set theory to prove the correctness of the class diagram models developed with this approach. Therefore, part of the next goal will concentrate on applying standard graph theorems to identify mathematical correctness in the structure of the model.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

K.S. and E. G. identified a new approach by simplifying the generalization/specialization semantics of the class diagram through the application of mathematical formality to usage of these class diagram concepts. K.S. and E. G. carried out the experiments to eliminate duplicity associated with UML class models. The paper was improved based on feedback and proofreading from E.G. All authors have read and agreed to the published version of the manuscript.

References

[1] Booch, G., Rumbaugh, J., & Jacobson, I. (1997). The unified modeling language, rational software corporation. Addison-Wesley, Indiana, USA.

[2] Object Modeling Group. Unified Modeling Language Specification. Version 2.5. October 2012.

[3] Gutwenger, C., Jünger, M., Klein, K., Kupke, J., Leipert, S., & Mutzel. P. (2003). A new approach for visualizing UML class Diagrams. *Proceedings of the ACM Symp. Software Visualization (SOFTVIS03)*,179-188.

[4] Grant, E. S., Whittle, J., & Chennamaneni, R. (2003). Checking program synthesizer input/output. *Generative Programming and Component Engineering (GPCE)*, Anaheim, CA, USA.

[5] Pfleeger, S. L., & Atlee, J. M. (2006). Software engineering: Theory and practice 4th Ed.. Prentice Hall.

[6] Varró, D. (2006). Model transformation by example. *Proceedings of the* I*nternational Conference on Model Driven Engineering Languages and Systems*.

[7] Sendall, S., & Wojtek, K. (2003). Model transformation: The heart and soul of model-driven software development. *IEEE Software 20*, 42-45.

[8] Potter, B., Sinclair, J., & Till, D. (1996). An introduction to formal specification and Z 2nd ed., Prentice Hall Europe.

[9] Chikofsky, E. J., & James, H. C. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, *7(1)*, 13-17.

[10] Taentzer, G. (2003). AGG: A graph transformation environment for modeling and validation of software. *International Workshop on Applications of Graph Transformations with Industrial Relevance*.

[11] Zhao, Y., Fan, Y., Bai, X., Wang, Y., Cai, H., & Ding, W. (2004). Towards formal verification of UML diagrams

based on graph transformation. *Proceedings of the IEEE International Conference on E-Commerce Technology for Dynamic E-Business*.

[12] Chama, W., Elmansouri, R., & Chaoui, A. (2012). Model checking and code generation for UML diagrams using graph transformation. *International Journal of Software Engineering & Applications*.

[13] Outair, A., Abdelouahid, L., & Mariam, T. (2014). Towards an automatic evaluation of UML class diagrams by graph transformation. *International Journal of Computer Applications*.

[14] Herout, P., & Premek, B. (2016). Uml-test application for automated validation of students' uml class diagram. *Proceedings of the 2016 IEEE 29th International Conference on Software Engineering Education and Training*.

[15] Modi, S., Hanan, A., Taher, & Hoger, M. (2021). A tool to automate student UML diagram evaluation. *Academic Journal of Nawroz University*, 189-198.

[16] Bian, W., Omar, A., & Jörg, K. (2019). Automated grading of class diagrams. *Proceedings of the 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*.

[17] O'Regan, G. (2017). Concise guide to formal methods: Theory, fundamentals and industry applications. *Undergraduate Topics in Computer Science Series*, Springer International Publishing.

[18] France, R., Andy, E., Kevin, L., & Bernhard, R. (1998). The UML as a formal modeling notation. *Computer Standards & Interfaces*, 325-334.

[19] Evans, A. S. (1998). Reasoning with UML class diagrams. *Proceedings of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*.

[20] Clark, T., & Evans, A. (1997). Foundations of the unified modeling language. *Proceedings of the 2nd Northern Formal Methods Workshop*. Ilkley, U.K.: Springer.

[21] Spivey, J. Michael, & Abrial, J. R. (1992). *The Z notation*. Hemel Hempstead: Prentice Hall.

[22] Cadoli, M., Diego, C., Giuseppe, D. G., & Toni, M. (2004). Finite satisfiability of UML class diagrams by constraint programming. *CSP Techniques with Immediate Application*.

[23] Malgouyres, H., & Gilles, M. (2006). A UML model consistency verification approach based on meta-modeling formalization. *Proceedings of the 2006 ACM Symposium on Applied Computing*.

[24] Pérez, B., & Ivan, P. (2019). Reasoning about UML/OCL class diagrams using constraint logic programming and formula. *Information Systems*, 152-177.

[25] Cabot, J., & Ernest, T. (2006). Incremental evaluation of OCL constraints. *Proceedings of the International Conference on Advanced Information Systems Engineering*.

[26] Cabot, J., & Robert, C., & Daniel, R. (2008). Verification of UML/OCL class diagrams using constraint programming. *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*.

[27] Cabot, J., Robert, C., & Daniel, R. (2007). UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*.

[28] Clarisó, R., Carlos, A. G., & Jordi, C. (2015). Towards domain refinement for UML/OCL bounded verification. *In SEFM 2015 Collocated Workshops*.

[29] Edwards, A. W. F. (2004). Cogwheels of the Mind: The story of Venn Diagrams 1st Ed., John Hopkins University Press, Baltimore, USA & London, UK.

[30] Glass, R. L. (1997). *The Software-Research Crisis, IEEE Software*, IEEE Computer Society Press, California, USA.