Global Journal of Computer Science and Technology: H
Information & Technology
Volume 23 Issue 1 Version 1.0 Year 2023
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Towards Verification of UML Class Models using Formal Specification Methods: A Review

By Kruti P. Shah & Emanuel S. Grant

University of North Dakota

Abstract- In today's world, many elements of our lives are being affected by software and for that we are in greater need of high-quality software. The Unified Modeling Language (UML) is considered the de facto standard for object-oriented software model development. UML class diagram plays an important role in the design and specification of software systems. A class diagram provides a static description 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. However, there are many concepts in the UML with imprecise semantics for that reason the models created may be incorrectly designed. Also, there are number of designers involved in the model designing process who are prone to making mistakes, which gives rise to potential conflicts, uncertainty, and ambiguity. The development of these models is a highly time-intensive process. Therefore, it is extremely important to check the correctness of these models and identify the problems in the early stage of the software development process.

Keywords: formal methods; model verification; MDE; UML models; UML class diagrams.

GJCST-H Classification: FOR Code: 090699

TOWARDSVERIFICATIONOFUMLCLASSMODELSUSINGFORMALSPECIFICATIONMETHODSAREVIEW

Towards Verification of UML Class Models using Formal Specification Methods: A Review

Kruti P. Shah [α] & Emanuel S. Grant [σ]

Abstract- In today's world, many elements of our lives are being affected by software and for that we are in greater need of high-quality software. The Unified Modeling Language (UML) is considered the de facto standard for object-oriented software model development. UML class diagram plays an important role in the design and specification of software systems. A class diagram provides a static description 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. However, there are many concepts in the UML with imprecise semantics for that reason the models created may be incorrectly designed. Also, there are number of designers involved in the model designing process who are prone to making mistakes, which gives rise to potential conflicts, uncertainty, and ambiguity. The development of these models is a highly time-intensive process. Therefore, it is extremely important to check the correctness of these models and identify the problems in the early stage of the software development process. Error detection and verification of these models at early stage may save costs and time of software development. Therefore, an integration of UML and formal methods is required to overcome such type of issues. Formal methods have proven effective in the development of safety critical systems. The purpose of this work is to provide an overview of formal specification methods (Z notation and OCL) for verifying the UML class model. This review will be helpful to understand current research trends and identify open issues or other areas for improvement in the domain of UML class model verification.

Keywords: formal methods; model verification; MDE; UML models; UML class diagrams.

I. Introduction

Graphical models of software systems are designed and developed in the initial phase of the Software Development Life Cycle (SDLC) [1]. A model is an abstract representation that is used to analyse and understand a different aspect of software system [2]. In Model-Driven Engineering (MDE), the software design model is considered a foundation of all development activities. Models in software engineering are used to elicit requirements, design the system, and develop the code of the proposed system.

In software engineering, it is essential and beneficial to design a model before the implementation. It provides an understandable view of the system and improves communication among technical developers and non-technical users. Along with that, the software design model provides identification of ambiguities and uncertainties at the initial level of SDLC with the help of model verification techniques [3,4].

Unified Modeling Language (UML) [2] is a widely used graphical modeling language, and it is extensively used in MDE. It is used to specify, simulate, and construct software system components. The UML has been adopted and standardized by the Object Modeling Group [5]. It has many static and dynamic models for dealing with different aspects of software.

The class model is an essential part of UML which performs a major role in analysis and design of software [5].This work considers the UML class diagram, which is the most fundamental and widely used among all UML models according to a survey presented in [6]. A Class Diagram provides a static description of system components. The key components of a class model are classes with their attributes and methods, hierarchy (generalization) class relationships, and associations (general, aggregation, and composition) [2, 7].

UML is considered the standard for object-oriented software model development that allows modeling of various aspects of complex systems [2, 7]. However, there are many concepts in the UML with imprecise semantics, which limit the use of the UML and reduce the quality of the UML models. Also, they lack a formal foundation. Therefore, model verification is not possible in them. Thus, developing technologies for the analysis and verification of UML models is significant to developers who use UML for system modeling.

The programming language code is developed with the reference of the design models in MDE, and defects and ambiguities in the model can implicitly transfer into the programming code, making it more difficult to determine and rectify. Also, the development of these models is a highly time-intensive process. Therefore, it is extremely important to check the correctness of these models and identify the problems in the early stage of the software design process.

Model verification ensures that the design model is unambiguous, correct, and bug-free. It essentially verifies the model's accuracy and guarantees that the model is consistent and acceptable. The ability to analyse and validate UML models is provided by

Author α σ: School of Electrical Engineering and Computer Science, University of North Dakota, Grand Forks, ND, USA.
e-mails: kruti.shah@ndus.edu, emanuel.grant@und.edu

formal specification methods [8]. Formal methods involve the use of a specification language and mathematical theories to design models. They enhance consistency, eliminate design flaws, and improve system reliability.

Despite the challenges that model complexity has introduced into MDE-based software development processes, as well as the benefits of using formal methods to verify software, there has been a lot of work done on applying formal methods and formal analysis techniques to ensure the model correctness.

This paper reviews the progress of some research articles on UML class model verification methods Z(zed) and Object Constraint Language (OCL) and directs future research in the area of formal specification language. The primary goal of this work is to provide a summary of approaches considered in selected articles, along with the quality of their results and conclusions. This review will be useful for researchers to understand the important open issues in existing methods and limitations that need to be addressed in the area of model correctness.

The remainder of the paper is organized as follows. Section 2 represents the review process including the research questions and the inclusion/exclusion criteria. Section 3 gives a brief theoretical background of UML class model along with the model transformation and formal methods to verify the correctness of UML models. Section 4discusses the studies and work done in the area of verification and correctness of UML class models using formal methods. Section 5 discusses the review summary and important open issues in the domain of formal specification methods followed by the conclusion.

II. Review Process

This section discusses the Research Questions followed by defining inclusion and exclusion criteria for the review.

a) *Research Questions*

This paper focuses on providing an analysis and comparison of the research initiatives done in the field of formal verification approaches mainly Z(zed) notation and Object Constraint Language (OCL). More precisely, we aim to answer the following research questions in this literature review:

RQ1: What is the importance of UML models and static CD models?

RQ2: What is the importance of model transformation and formal specification methods?

RQ3: Which model defects have been undertaken in proposed approach?

RQ4: Is a verification approach supported by the tool?

RQ5: What are the deficiencies associated with the selected formal approach?

b) *Inclusion/Exclusion Criteria*

In this section, we defined inclusion/exclusion criteria to determine the related works. The inclusion criteria focus on: 1)studies related to the verification of UML class model using formal methods Z and OCL and 2) paper published in English. On the other hand, We exclude the formal verification studies that are related to dynamic UML models. Based on the inclusion/exclusion criteria, I have selected following studies that are related to the Z-notation and OCL for this review.

Table 1: Selected studies for this survey

Study	Method	Reference Title
S1	Z-notation	1) The UML as a formal modeling notation 2) Reasoning with UML class diagrams 3) Foundations of the unified modeling language 4) The Z notation Manual
S2	OCL	1) Finite satisfiability of UML class diagrams by constraint programming 2) A UML model consistency verification approach based on meta-modeling formalization 3) Reasoning about UML/OCL class diagrams using constraint logic programming and formula 4) Incremental verification of UML/OCL models 5) Verification of UML/OCL class diagrams using constraint programming 6) UML to CSP: A tool for the formal verification of UML/OCL models using constraint programming 7) Towards domain refinement for UML/OCL bounded verification

III. Theoretical Background

This section covers some of the theories and prior work in the area of UML models and various aspects of UML class diagrams along with the description of the requirement of model transformation and formal specification methods to verify the correctness of such models.

a) *Unified Modeling Language (UML)*

UML [2,7] has been widely accepted as the standard language for modeling and documenting software systems. Their 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 diagram forms to describe particular aspects of software artifacts. These diagram structures can be divided into two categories static or 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. The dynamic view reflects the changes related to the internal states of individual objects and changes in the system's overall state. This view can be represented by sequence, activity, and state chart diagrams.

i. *UML Class Diagram*

The UML class diagrams are used to represent the static structure of system components [2,7]. It describes the system structure in terms of classes, attributes, and constraints imposed on classes (operations) and their inter-relationships.This work focuses on the 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, set of attributes, and 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.

ii. *UML Association (Aggregation, Association, Composition, generalization)*

There are some rules and requirements for combining the classes to construct partial or complete UML class models.

Association It can be depicted as bi-directional or unidirectional. The association lines indicate the possible relationship between the class entities [9]. An association represents attributes and objects from the related classes, such as the relationship between class A and class C seenin Fig. 1.

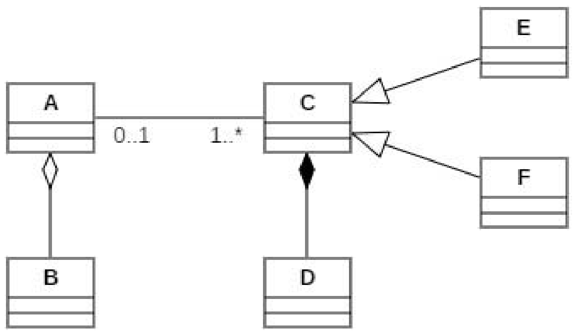

Figure 1: UML Class Diagram

Association ends can be annotated with labels, known as association end names and multiplicities. For example, multiplicity can be expressed as specific numbers, ranges of numbers, or unlimited 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). It includes or owns instances of the class (contained class) at the other end of the association [9] (e.g., the relationship between class A and B in Figure 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 [9]; if an instance of the container class is deleted, the instances of the contained class are also deleted. Figure 1 shows class C, the container class, and class D, the contained class. It is represented as an association with a black diamond.

Generalization—▷A generalization is represented by an association with a triangle on one end represents, where the class at the triangle end of the association is the parent class of the classes at the other ends of the association, called subclasses [9]. 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).

b) *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 [10].

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 [11].

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 [11]. 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.

c) *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 primary goal of our review is to compare two of these formal specification approaches i.e., Z notation and OCL.

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.

The importance of formal languages increases in real-time safety critical systems where the primary concern is reliability and performance of the system. There is decent amount of work done in terms of defining and specifying formal languages for software systems and UML models, with some being accepted by the industry, such as Z, OCL, VDM, B, Alloy, etc. As each language has its own pros and cons, this survey compares two languages Z and OCL that can be utilize for verifying real-time safety critical systems.

i. *Z-notation*

The Z notation [12]-[15] is based on first-order logic and typed set theory. A schema i.e., a component of Z notation that describes the state and operations of a specification. A schema is a collection of variable declarations accompanied by a set of predicates that constrain the variable's possible values.

ii. *Object Constraint Language (OCL)*

The Object Constraint Language (OCL) [16]-[22] is a constraint expression language for object-oriented languages and other modeling artifacts. OCL is a component of the Unified Modeling Language (UML) that plays a key role in the software lifecycle's analysis phase. For a detailed and unambiguous specification, traditional graphical models, such as class models, are insufficient. Therefore, We require to add some more constraints to the objects to resolve those issues. However, the classic formal method requires a significant knowledge of mathematics, making it difficult for the average business or system modeler to employ. OCL has been designed to bridge this gap. It was created by IBM's Insurance group as a business modeling language.

IV. Literature Review

a) *Z notation*

The Z notation is used in the first research [S1, 12-15] 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 meta model in S1. They claimed that the formal foundation provides a number of benefits, including transparency, extendability, consistency testing, refinement, and proof [12, 13].

They have defined a compositional schema with multiple subschema a as 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 [12]: 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 [12] 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 [14] 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 [15], 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 [15].

b) Object Constraint Language (OCL)

In the second study [S2, 16-22], 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 [16]. 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 [17] employed Constraint Logic Programming (CLP). They used CLP clauses to translate the UML class model, metamodel, and meta-metamodel [17]. In this approach, c 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 [18] 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. [19] 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 [19]. 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. [20] 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 [21] 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 UML to CSP tool in [21]. 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. [22] expanded on their previous work [20], 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 [22]. 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.

V. Review Summary and Conclusion

Software design models play an important part in modern software development. They are useful for more than just documentation; they are also used for analysis, design, testing, and even code development using an automated transformation technique. The transformation technique allows existing software artifacts to be reused automatically. However, it has several flaws, such as the fact that model flaws are automatically transmitted to the changed model through the transformation. These flaws are difficult to detect and correct. Model verification appears to be a viable solution to the problem.

The verification of the UML class model is essential for assuring model quality prior to transformation. Verification of the UML class model through formal notation has been discussed in several studies. In this review, we discussed prior works in the field of formal specification specially related to Z and OCL methods. We presented a comparison of these formal methods in Table 2 based on the analysis of studies [12]-[22]. This comparison is performed based on the features like support for UML features, Tool support, feedback for the user, and the efficiency of the methods and verification tool. Both the methods provide support for association, aggregation, and generalization relationships and do not support the features like dependency relationships (aggregation and composition) and x or constraint. Z notation is supported by Z word and Z/Eves verification tools. USE and UML to CSP tools are capable of working with OCL. Both of these tools support semi-automatic transformation.

Table 2: Comparison of Z and OCL formal methods

Method	Support for UML Features	Tool Support	Feedback to user	Efficiency
Z	Association, Generalization, Multiplicity Constraints	Z Word, Z/Eves	Error: Does not provide meaningful feedback Successful: message in textual form on a pop-up window	Not efficient with large or complex UML class models
OCL	Association, Association Classes, Generalization, Multiplicity Constraints	USE Tool UMLtoCSP	Error: Does not provide meaningful feedback Successful: object model	Not efficient with large or complex UML class models

Both the tools (Z word and USE) provide feedback to users in order to notify them of the verification process's outcome. Z word provides the successful message in textual form on a pop-up window. In case of USE tool, if the verification process ends successfully it is complemented by a sample object model. This sample object model acts as the proof of the verification. When the verification process does not succeed, the Z Word and USE tools can display some hints in textual form on a window. This can help model designers in identifying the reasons for the failure and adjusting the model accordingly.

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. Overall, We can say that the existing verification tools, apart from being certainly limited in size, is in some cases targeted at a very limited or specific audience.

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.

Unfortunately, none of the verification tools examined in this study performs well in terms of achieving the verification requirements. These tools and methods in general do not integrate well and have been developed to conduct verification apart from the rest of the activities that characterize a model designer's work. It forces users to switch between model editors and verification tools to check for errors every time models are refined or improved, usually with little or no hint on where to apply fixes if the verification fails.

To conclude this, in my 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, x or 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.

References Références Referencias

1. Davis, Alan M., Edward H. Bersoff, and Edward R. Comer. "A strategy for comparing alternative software development life cycle models." IEEE Transactions on software Engineering 14, no. 10 (1988): 1453-1461.
2. G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language, Rational Software Corporation, Addison-Wesley, Indiana, USA, 1997.
3. Mens, Tom, and Pieter Van Gorp. "A taxonomy of model transformation." Electronic notes in theoretical computer science 152 (2006): 125-142.
4. Meedeniya, Dulani Apeksha. "Correct model-to-model transformation for formal verification." PhD diss., University of St Andrews, 2013.
5. Object Modeling Group. Unified Modeling Language Specification. Version 2.5. October 2012.
6. Dobing, Brian, and Jeffrey Parsons. "How UML is used." Communications of the ACM 49, no. 5 (2006): 109-113.
7. Rumbaugh, J.; Jacobson, I.; Booch, G. The Unified Modeling Language Reference Manual; Pearson Higher Education: Hoboken, NJ, USA, 2004.
8. Lano, A. Evans R. France K., and B. Rumpe. "The UML as a Formal Modeling Notation." Computer Standards and Interfaces 19 (1998): 325-334.
9. C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. "A New Approach For Visualizing UML Class Diagrams." Proc ACM Symp. Software Visualization (SOFTVIS03), June 2003, Association for Computing Machinery, pp. 179-188.
10. Varró, Dániel. "Model transformation by example." In International Conference on Model Driven Engineering Languages and Systems, pp. 410-424. Springer, Berlin, Heidelberg, 2006.
11. Sendall, Shane, and Wojtek Kozaczynski. "Model transformation: The heart and soul of model-driven software development." IEEE software 20, no. 5 (2003): 42-45.
12. France, Robert, Andy Evans, Kevin Lano, and Bernhard Rumpe. "The UML as a formal modeling notation." Computer Standards & Interfaces 19, no. 7 (1998): 325-334.
13. Evans, Andy S. "Reasoning with UML class diagrams." In Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, pp. 102-113. IEEE, 1998.
14. T. Clark and A. Evans, ``Foundations of the unified modeling language," in Proc. 2nd Northern Formal Methods Workshop. Ilkley, U.K.: Springer, Jul. 1997, pp. 1-15.
15. Spivey, J. Michael, and J. R. Abrial. The Z notation. Vol. 29. Hemel Hempstead: Prentice Hall, 1992.
16. Cadoli, Marco, Diego Calvanese, Giuseppe De Giacomo, and Toni Mancini. "Finite satisfiability of UML class diagrams by Constraint Programming." CSP Techniques with Immediate Application (CSPIA) 2 (2004): 2-16.
17. Malgouyres, Hugues, and Gilles Motet. "A UML model consistency verification approach based on meta-modeling formalization." In Proceedings of the 2006 ACM symposium on Applied computing, pp. 1804-1809. 2006.
18. Pérez, Beatriz, and Ivan Porres. "Reasoning about UML/OCL class diagrams using constraint logic programming and formula." Information Systems 81 (2019): 152-177.
19. Cabot, Jordi, and Ernest Teniente. "Incremental evaluation of OCL constraints." In International Conference on Advanced Information Systems Engineering, pp. 81-95. Springer, Berlin, Heidelberg, 2006.
20. Cabot, Jordi, Robert Claris, and Daniel Riera. "Verification of UML/OCL class diagrams using constraint programming." In 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, pp. 73-80. IEEE, 2008.
21. Cabot, Jordi, Robert Clarisó, and Daniel Riera. "UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming." In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 547-548. 2007.
22. Clarisó, Robert, Carlos A. González, and Jordi Cabot. "Towards domain refinement for UML/OCL bounded verification." In SEFM 2015 Collocated Workshops, pp. 108-114. Springer, Cham, 2015.