applied sciences

MDPI

Article

UCLAONT: Ontology-Based UML Class Models Verification Tool

Adel Rajab [1], Abdul Hafeez [2], Asadullah Shaikh [1], Abdullah Alghamdi [1], Mana Saleh Al Reshan [1], Mohammed Hamdi [1] and Khairan Rajab [1,*]

[1] College of Computer Science and Information Systems, Najran University, Najran 61441, Saudi Arabia; adrajab@nu.edu.sa (A.R.); asshaikh@nu.edu.sa (A.S.); aalghamdi@nu.edu.sa (A.A.); msalreshan@nu.edu.sa (M.S.A.R.); mahamdi@nu.edu.sa (M.H.)
[2] Department of Software Engineering, SMI University, Karachi 76400, Pakistan; ahkhan@smiu.edu.pk
* Correspondence: kdrajab@nu.edu.sa

Abstract: The software design model performs an important role in modern software engineering methods. Especially in Model-Driven Engineering (MDE), it is treated as an essential asset of software development; even programming language code is produced by the models. If the model has errors, then they can propagate into the code. Model verification tools check the presence of errors in the model. This paper shows how a UML class model verification tool has been built to support complex models and unsupported elements such as XOR constraints and dependency relationships. This tool uses ontology for verifying the UML class model. It takes a class model in XMI format and generates the OWL file. Performs verification of model in two steps: (1) uses the ontology-based algorithm to verify association multiplicity constraints; and (2) uses ontology reasoner for the verification of XOR constraints and dependency relationships. The results show the proposed tool improves the verification efficiency and supports the verification of UML class model elements that have not been supported by any existing tool.

Keywords: ontology; model verification; class model; verification tool; MDE

Citation: Rajab, A.; Hafeez, A.; Shaikh, A.; Alghamdi, A.; Al Reshan, M.S.; Hamdi, M.; Rajab, K. UCLAONT: Ontology-Based UML Class Models Verification Tool. *Appl. Sci.* **2022**, *12*, 1397. https://doi.org/10.3390/app12031397

Academic Editor: Jinho Kim

Received: 31 August 2021
Accepted: 25 January 2022
Published: 28 January 2022

1. Introduction

A model is an abstract representation that is used to analyze and comprehend a system [1]. In the engineering world, models are used in almost every discipline, such as the house map in civil engineering and circuit diagram in electronic engineering. These models are used to understand the characteristics of the system [2]. In software engineering, design models play a vital role in software development. In modern software engineering methodologies, they are used to elicit requirements, design the system, and generate the code.

Software design models are a formal description of the structures and behaviors of the system. The discourse is the complete software design, including functionality, architecture, collaboration, user interfaces, and interaction with other software [3]. In software engineering, designing a model before the implementation is very beneficial. It provides an understandable view of the system and improves communication among all stakeholders. Furthermore, the software design model provides early identification of incompleteness and inconsistencies in the underdeveloped system with the help of model verification techniques [4,5].

Current software design modeling techniques such as Unified Modelling Language (UML) and Systems Modeling Language (SysML) are robust and cover all aspects of software development. However, they do not have a formal foundation. Therefore, verification of the model is not possible in them. On the other side, formal modeling methods provide the capability of analysis. A formal model mathematically represents the software specification and provides an automated reasoning facility [6–8]. They eliminate ambiguities, design faults and improve system reliability [9]. However, they are complex and

inspired by mathematics [10]. UML is a modeling language that is especially involved in the specification and visualization of the object-oriented system. Initially, Rational software developed the UML, but now Object Modeling Groups (OMG) managing it [11]. In the UML, an underdeveloped software system is represented by a set of different models, and each model focuses on a different facet of the software.

The UML class model is the most essential element of UML. According to a survey presented in [12], it is the most frequently used UML model. It is also a vital ingredient of the MDE process [13–15]. The main elements of a class model are classes and different types of relationships such as dependency, association, and generalization [16]. Association and generalization are also dependency relationships; however, they have specific semantics [16]. These three relationships (dependency, association, generalization) are the basic building block of UML and object-oriented modeling [16].

In modern software development methodology like Model-Driven Engineering (MDE), the software design model is considered a nucleus of all development activities and is recognized as a core element rather than a traditional programming language code [3,4,17,18]. In Model-Driven Software Engineering, the programming language code is produced from the design model automatically, and defects of the model can implicitly transfer in the programming code, which is harder to determine and repair. Model verification is a feasible solution to this problem.

Model verification ensures that the model will be built without errors. It makes sure that the model must have some important correctness features such as satisfiability and consistency [19]. Model is considered incorrect when it does not satisfy any correctness features, and when it satisfies all correctness features, it is considered correct [19].

Satisfiability, consistency, and well-formedness are the most fundamental correctness proprieties [19–21]. The consistency verifies whether the model elements are consistent with the declaration, whereas well-formedness verifies whether a model is a valid instance of its meta-model [19,20]. However, both of them only verify the initial level of syntax weaknesses and do not concern the model's semantic correctness. Semantic correctness concerns the constraints which are specified in the model graphically such as associations, dependencies, and generalizations, or textually defined through constraint language such as Object Constraint Language (OCL).

The most fundamental semantic correctness property of the UML class model is satisfiability [19]. Other important correctness properties which are verified and come under the umbrella of satisfiability are strong satisfiability, weak satisfiability, and class liveliness [22]. Strong satisfiability checks whether a model can instantiate successfully in which one instance of each element successfully populates. Weak satisfiability checks whether at least one or more elements of the model can be instantiated successfully. Class liveliness checks whether a class can populate successfully. The problem addressed in this work is twofold and can be stated as follows:

Firstly, current UML class model verification methods are sufficiently good to check correctness. However, they do not focus on some fundamental class model elements such as dependency relationships and XOR constraints. With few exceptions in which the XOR constraint is indirectly supported by some existing methods through OCL, however, OCL has some limitations. For example, UML specification does not restrict constraint language, and according to the UML specification, constraints can be defined through formal languages, informal languages (JAVA, C#), and natural language [23]. Most of the Computer-Aided Software Engineering (CASE) tools do not support OCL or provide limited support because commercial CASE tools do not see a large market of OCL [24].

Secondly, the problem faced by the current verification methods is scalability due to its high computational complexity [13,25]. When they deal with large and complex models, they consume enormous computational resources and time [14]. In [19], authors also identified different research directions for future UML class diagram verification methods, in which one is search space reduction for dealing with scalability issues. Therefore, there is

a need for UML class model verification methods that efficiently verify large and complex models within a reasonable time and with minimum computational resources.

Previously, we proposed ontology-based transformation and verification of UML class model unsupported elements, that is, XOR constraints and dependency relationships. These transformations map XOR and dependency relationships to an ontology for verification of various correctness properties such as satisfiability, consistency, and consequences. The proposed method verifies a large UML class model in an acceptable time [26]. In this paper, we extend our work in multiple directions. Firstly, it presents the complete transformation mechanism of UML's XMI to ontology. Secondly, it demonstrates the implementation details (Pseudocode) of the verification method proposed in the previous work. Finally, it presents the complete transformation and the verification model. Furthermore, this paper explores the various components of the tool and algorithms which have been used in the Ontology-Based Verification method. Additionally, it shows the performance of the tool in the real world for large and complex UML class models. In total, this paper presents the software implementation of our previously published works [26–29].

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 focuses on the running example along with the details of ontology-based finite satisfiability. Section 4 explores the tool architecture with all details of transformation rules. Section 5 describes the experimental results measuring verification time as compared to other tools. Finally, Section 6 provides the conclusions and future directions.

2. Related Work

The UML class model provides a graphical modeling notation without any formal foundation [30]. The well-formedness rules have been specified in the meta-model without any formal proof [31]. Therefore, different formal methods have been used to formalize and verify meta-model and well-formedness in the initial research work [32,33]. Furthermore, they also performed different analyses on the UML class models, such as diagrammatical transformation analysis, intersection, and refinement analysis [33,34]. However, most recent works focus on the consistency and satisfiability of the UML class model [24,35–40].

Ledang et al. [41] presented a tool ArgoUML+B on ArgoUML. This tool transformed classes, class attributes, and class operations into B Machine and transformed OCL constraint into B's method. In ArgoUML+B, the UML class model is inputted as XMI and transformed into the B specification. Finally, this tool uses B Prover for checking the consistency of the UML class model. Marcano et al. [42] build a tool called OCL2B, which performs analysis on the UML class model and OCL. In the tool, the transformation process performs in two stages. (1) an abstract machine is declared, representing the class structure and the associations, and (2) abstract machines are made for all classes. OCL2B uses OCAML language for transformation rules. Maraee et al. [37] developed a tool called FiniteSatUSE, which uses the proposed algorithm FiniteSat for verification of generalization relationships of the class model. This tool performs bounded verification of generalization set constraint and analysis on different types of generalization, for example, tree structure, acyclic structure, and graph structure through linear inequalities system.

Cabot et al. [43] proposed a tool called UMLtoCSP which uses Constraints Satisfaction Problem (CSP) for representation of UML class diagram. In this tool, all UML class model elements are represented through the set of variables, domains, and constraints of CSP, and verification is performed in 2 stages. (1) cardinality variables for every instance of classes, associations, domain, and entire constraints are defined. This step aims to allocate legitimate values to all CSP variables, and if it is not possible, then CSP is declared unsatisfiable. (2) an instance model is built through the value of cardinality variables. UMLtoCSP takes a UML class model in the XML Meta-Model Interchange (XMI) format and OCL in a separate text file. The XMI is parsed through the MDR, and the Dresden OCL Toolkit processes OCL constraints. Anastasakis et al. [44] built a tool called UML2Alloy. It formalized the class model and OCL into the specification of Alloy and used Alloy Analyzer for variation. Maozi et al. [34] presented a plug-in CD2Alloy for ECLIPSE. It

mapped advanced features of the class model through a combination of the basic construct of Alloy. It supports different analyses on the UML class model, such as the intersection and refinement analysis. It uses the FreeMarker template for making transformation rules.

Currently, ontology is extensively being used in software development practices, and many researchers have used it for the specification and verification of numerous software engineering objects and different UML models.

Nguyen et al. [45] combined USE-CASE modeling and goal-oriented techniques and presented a verification framework through ontology for checking inconsistency, incorrectness, and incompleteness. The authors also created a tool called "GUITAR", which takes input requirements in a text format and transforms them into ontology. The tool performs automatic reasoning through built-in reasoners and generates comprehensive feedback about the problems. Corea et al. [46] represented the business in the ontology and presented an ontology-based method for verifying business processes. In this method, business rules are transformed in the logic program, and reasoning is performed to identify model elements that violate the rules. Fellmann et al. [47] proposed an ontology-based technique for transformation and verification of business process model. This work divided the verification task into two steps. In the first step, different model elements are transformed into the ontology, and constraints of the model are represented through the rules. In the second step, the ontology model and rules are tested through the built-in reasoner.

Sun et al. [48] proposed the transformation of software architecture into Web Ontology Language (OWL) and used Semantic Web Rule Language (SWRL) for representing the dynamic communication (constraints). Kezadri et al. [49] presented an ontology-based transformation and verification of the behavioral model. They also proposed transforming various verification and validation terms, elements, and relationships into the ontology. He et al. [31] proposed ontology-based verification of UML behavioral models. The behavior model is divided into static elements specified through ontology and dynamics elements specified by DL-safe rules in this work. Dilo et al. [50] proposed the difference between UML and Web Ontology Language (OWL) and presented many common elements such as classes, attributes, and relationships. They also identified the difference between UML and OWL, such as UML having many relationships (association, aggregation, and composition). On the other side, OWL only has an object property.

Lastly, they presented that OWL and UML are compatible with each other. Bahaj et al. [51] presented a different transformation procedure for the class model to ontology. They transformed encapsulation and aggregation/composition into Object Property. Belghiat et al. [52] proposed a graph-based transformation of the UML class Model into ontology. Parreiras et al. [53] combined UML with OWL-DL for representing the software model. They integrated the MOF meta-model as the backbone for both UML and OWL.

R. Clariso et al. [54] proposed incremental verification of the UML class model. The significant feature of the proposed technique is the use of valid instances of the UML class model as certificates of satisfiability. The proposed technique implemented the UML-based Specification Environment (USE) Tool. Abbas, M et al. [55] present FoCaLiZe development environment for specification and verification of UML class diagram. They generate FoCaLiZe specification of multiple inheritances, dependency template, template binding, and the navigation of OCL contained in the FoCaLiZe specification classes are transformed into species, properties of a class into getter functions, operations into signatures, and OCL constraints into species properties. FoCaLiZe uses Zenon theorem Prover for specification verification. Perez [56] proposed a framework for reasoning on the UML class and the OCL model through Constraint Programming Logic (CPL). They translate the UML class model in constraint satisfaction problems and performed reasoning through the model-finding formula and developed an Eclipse plug-in. the plug-in automatically translates the class diagram into the CPL formula.

Current verification methods of the UML class model are capable of checking correctness. However, they do support some fundamental constraints (such as XOR dependencies and XOR). Another problem faced by the existing methods is scalability, due to high com-

putational complexity when dealing with large and complex models verification methods consume enormous computational resources and time. Various existing research works also pointed out different research avenues for future UML class diagram verification methods in which most important is search space reduction for dealing with scalability issues. Therefore, there is a need of UML class model verification methods that efficiently verify large and complex models within a reasonable time and with minimum computational resources.

3. Running Example

Throughout the paper, we will use a running example, simple enough to illustrate the UCLAONT concepts and mechanisms concisely. This example represents the entire verification process of the ontology-based verification process. However, faulty model and its verification through ontology has been discussed in our previous work [27].

Figure 1 shows the running example "Monopoly Game", which will demonstrate the functionality of UCLAONT. Monopoly is a board game in which players roll dice to move around the game board and try to buy various properties. Players gather rent from other players to drive them into bankruptcy.

The "Model Monopoly" has six classes, eight associations, and two associations between "player" and "Unowned property" are annotated with the XOR constraint. Monopoly Game class is a central class which has association relationships with the three classes Die, Players, and Board. The Plays association between Player and Monopoly Game specifies that two to eight players can play the game together. The game will be played with two dice is specified through Played with the association between Die and Monopoly Game. The game will be played on one Board is specified by the association Played On. The Board has 40 Squares as specified by the Contains association between Board and Square classes. Zero to eight Pieces can be placed on Square is specified through Is-On association. The Player can either Buy or Auction the property but cannot perform both actions because the associations are marks as XOR.

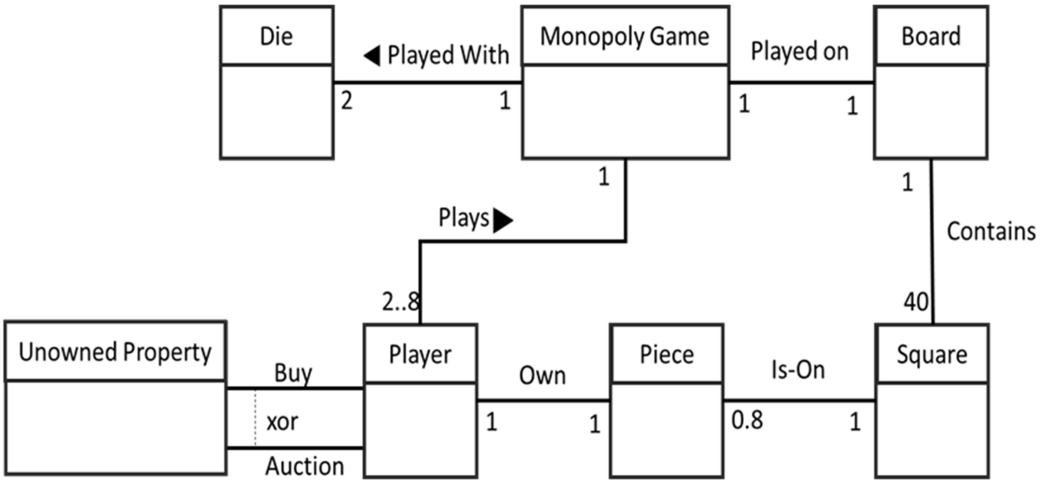

Figure 1. Running Example of Monopoly Game.

Ontology-Based Finite Satisfiability

There are two techniques for verifying satisfiability of static model: (1) linear inequalities; and (2) detection graph. In the first technique, a class model is transformed into linear inequalities, and the satisfiability problem is solved by finding the solution of inequalities. In the second technique, a class model is converted into a directed graph, and satisfiability is checked through the detection of the critical cycle. Ontology is also based on graph-theoretic structure and has concepts like vertices and edges of the graph. In this work ontology-based technique is used for finding critical cycles in the ontology graph of the class model. Figure 2 shows how an ontology graph of Figure 1 (Monopoly Game class model) will be built.

In graph-based representation, many cycles are not important for checking the class model's satisfiability, such as balance or greater. In the balance cycle, the same quantities are involved for the division and multiplication for calculating cycle weight. Therefore, quantity 1 is always produced. In the greater cycle, a smaller quantity for the division and a larger quantity is involved for multiplication. Therefore, a quantity greater than 1 is always produced. Both balance and greater cycles are not crucial for determining satisfiability. Therefore, they should not be part of the search. Furthermore, one critical cycle is enough for proving the unsatisfiability of the class model. As per the above investigation, this work proposes an ontology-based technique that minimizes the search space for finding a critical cycle in the ontology graph of the ontology-based representation of the UML class model. In this proposed tool, the object property is used for traversing the graph. In the proposed method, an arbitrary object is selected for traversing and step forward to the next object property until and unless the next selected object property range becomes equal to the domain of the first selected object property where the traversing was started. Further details can be found in [27].

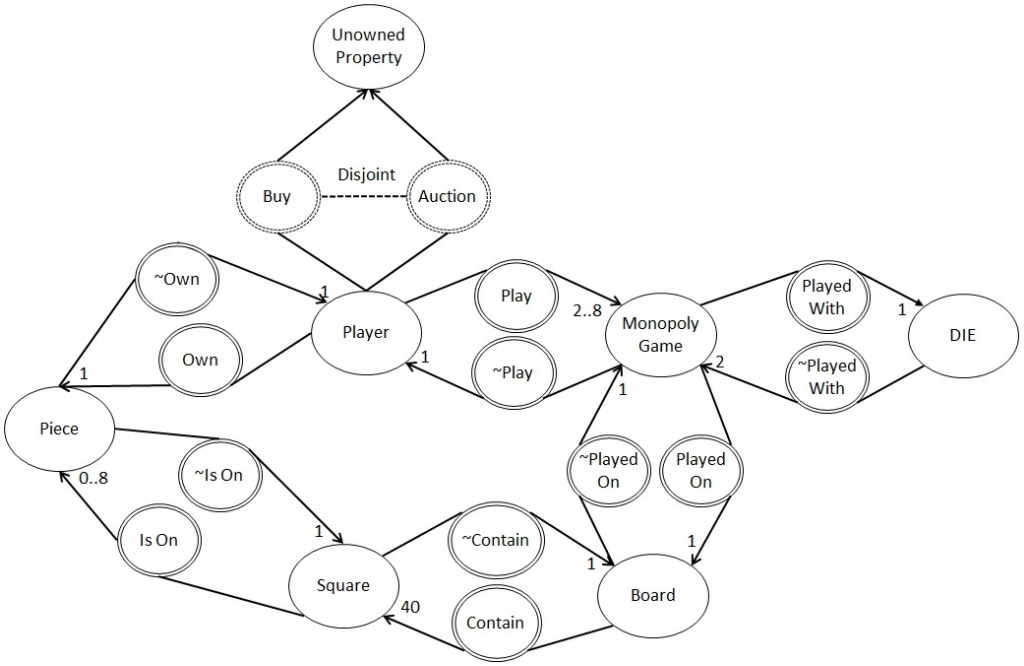

Figure 2. Ontology Graph of Monopoly Game Class Model.

UML classes connect to each other with different relationships; one of them is the dependency relationship. The dependency relationships specify the object of class effects on the object of another class such as in initiate dependency when an object of a class is created, it also creates an object of another class. The dependency relationships are not only used in the class diagram they can also be used in package diagrams, component diagrams, and so forth. The dependency relationships which impact the correctness of the class model only those are considered in this work such as create/Initiate, drive, call and use. In the proposed solution, the dependency relationship is transformed into object property and some additional restrictions are also applied on them such as use and calls are annotated as transitive, and drive and create/initiate are annotated as transitive and asymmetric. Sometimes a class model can become incorrect due to some concealed aspects of the dependency such as a class model presented in Figure 3. According to the model, Class A initiates the object of Class B, when the Class B object is initiated, it initiates the object of Class C. Furthermore, Class C has a generalization relationship with Class D, and the Class C object will also be considered a Class D object due to inheritance. ultimately, there will be a cycle of objects initialization that will never end and the model will not be finitely satisfiable.

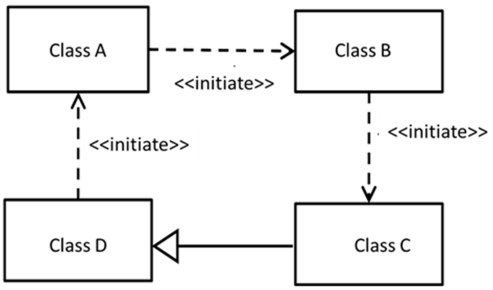

Figure 3. Infinitely Unsatisfiable Dependency Relationships.

4. UCLAONT Architecture

There are many tools available for the verification of the UML class model. However, none of them focus on XOR constraints and dependency relationships. They also face scalability issues: their performance goes down when they deal with large and complex models. UCLAONT tool uses the ontology-based approach to verify large and complex association constraints, XOR constraint and decadency constraint of the class model. Additionally, it provides the formalization and verification of various XOR constraints and different types of dependency relationships. The implemented approach in UCLAONT is very prompt and can check the correctness of large models within a few milliseconds. The tool has been implemented in Apache JENA, which is an open-source JAVA-based API. JENA is used for extracting and writing OWL and RDF. It has many inference engines and also provides support for many third-party inference engines. The core of JENA is a JAVA library that manipulates the ontology graph.

The proposed tool takes the UML class model in the XML Metadata Interchange (XMI) format, that is, XMI 2.41. The XMI provides the facility of exchanging UML models among the CASE tools in XML format. It provides a common format of UML models for sharing them among different CASE tools. The architecture of the prototype tool is shown in Figure 4. The UML class model in the XMI format is given as an input to the tool. The transformation component is responsible for transforming the UML class model into the ontology, so the proposed method can be implemented. The reasoning component is in charge of the verification of the UML class model. Finally, the feedback component generates feedback on the verification result.

4.1. Transformation Component

The transformation component translates the UML class model into the ontology. The transformation component contains the XLST templates, which map a given UML class model elements to the corresponding ontology elements, and used Saxon 9 XSLT parser to generate ontology TURTLE format. Figure 5 depicts the outline of the model transformation process within the UCLAONT tool.

4.1.1. Transformation Rules

Rules are developed as eXtensible Stylesheet Language Transformations (XSLT). XSLT is a language that transformed XML content into another Format, Such as HTML, RDF, OWL. In the transformation rules, we transformed the UML XMI file into an OWL file (Turtle Format). Other transformation techniques can be used for transformation, but XSLT is very powerful than DOM and SAX. Its templates are based on XPath, which are very powerful in term of performance to process XML documents. We used saxon9 to process the XSLT because its engine supports standard Java application programming interfaces and supports XSLT version 3.0.

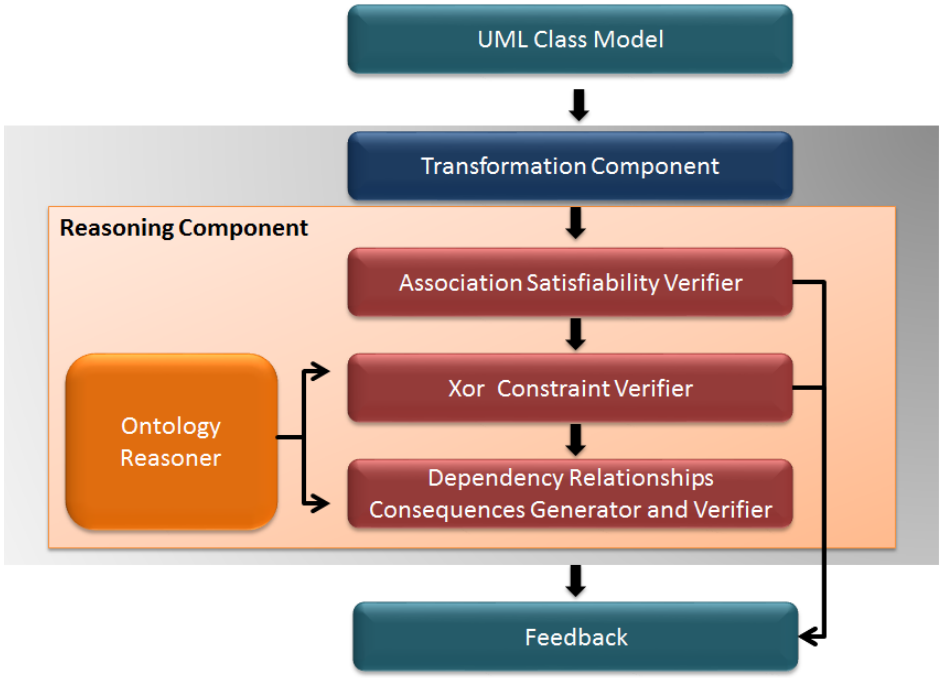

Figure 4. Architecture of UCLAONT.

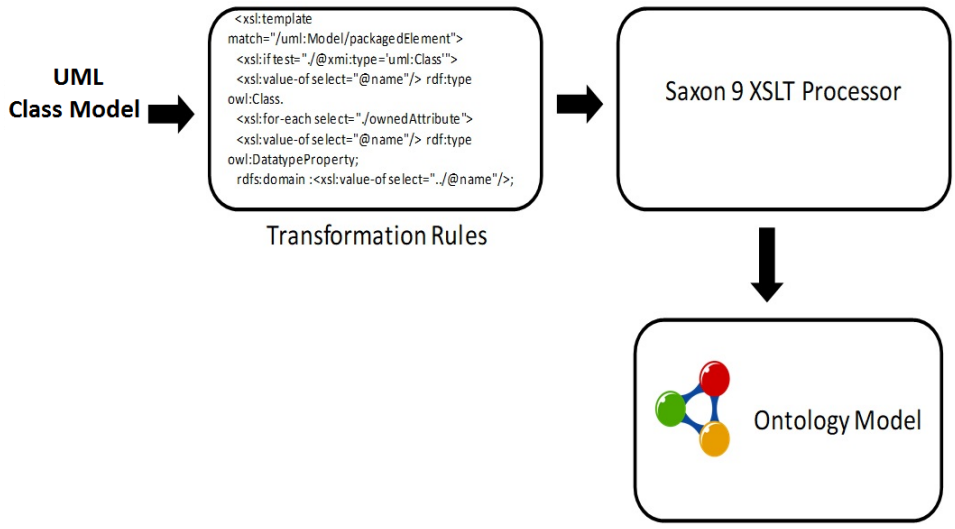

Figure 5. Model Transformation Process of UCLAONT.

4.1.2. Class and Attributes

In the proposed tool, UML classes are transformed into ontology classes. In XMI, we filter xmi:type uml:Class for classes and transformed it to ***rdf:type owl:Class***. The attributes of a class are declared as datatype properties of OWL and make the respected class a ***domain*** of the property and datatype set as a ***rang***.

```
Rules (XSLT)
<xsl:if test="./@xmi:type='uml:Class'">
:<xsl:value-of select="@name"/> rdf:type owl:Class.
</xsl:if>

Example (XMI)
<packagedElement xmi:type="uml:Class"
xmi:id="_0bvU1IwREeeHI8ZOY9KM8g" name="Player"/>

Output (OWL)
:Player rdf:type owl:Class.
```

4.1.3. Generalization Relationship

The generalization relation of the UML class diagram transformed into the subClass constraints of OWL. In XMI, we filter generalization keywords for generalization relationships. Then the key function is used for searching parent class.

```
Rules (XSLT)
<xsl:for-each select="./generalization">
:<xsl:value-of select="../@name"/> rdfs:subClassOf
:<xsl:value-of select="key('Classid',@general)/@name"/>.
</xsl:for-each>

Example (XMI)
<packagedElement xmi:type="uml:Class" xmi:id="_LxECg" name="A"/>
<packagedElement xmi:type="uml:Class" xmi:id="_LxECh" name="B">
<generalization xmi:type="uml:Generalization" xmi:id="_LxEC"
general="_LxECg"/>
</packagedElement>

Output (OWL)
:A rdf:type owl:Class .
:B rdf:type owl:Class ;
rdfs:subClassOf :A
```

4.1.4. Association

The association relationship is transformed into the object property of OWL, and respected classes appear as ***domain*** and ***range***. In XMI, we filter the ***UML: Association*** keyword for the association. We defined a customized function in XSLT for the identification of class positions in the association. If the class appeared in position one, it appeared as ***domain***, and the other class appeared as a ***range***. Additionally, another object property is added as an inverse of the declared property due to maintained two-way navigation between classes.

```
Rules (XSLT)
<xsl:if test="./@xmi:type='uml:Association'">
<xsl:choose>
<xsl:when test="mf:check(./ownedEnd/@xmi:id,../ownedRule/
@constrainedElement)"> </xsl:when>
<xsl:otherwise>
:<xsl:value-of select="@name"/> rdf:type owl:ObjectProperty;
<xsl:for-each select="./ownedEnd">
<xsl:if test="position()=1">
rdfs:domain :<xsl:value-of select="key('Classid',@type)/@name"/>;
</xsl:if>
<xsl:if test="position()!=1">
rdfs:range :<xsl:value-of select="key('Classid',@type)/@name"/>.
</xsl:if>
</xsl:for-each>
```

```
:inv-<xsl:value-of select="@name"/> rdf:type owl:ObjectProperty;
owl:inverseOf :<xsl:value-of select="@name"/>.

Example (XMI)
<packagedElement xmi:type="uml:Class" xmi:id="_C" name="Player"/>
<packagedElement xmi:type="uml:Association" xmi:id="_D"
name="Plays" memberEnd="_CE">
<ownedEnd xmi:type="uml:Property" xmi:id="_CE" type="_CJ"
association="_D"/>
<ownedEnd xmi:type="uml:Property" xmi:id="_Cc"
type="_Cf" association="_D"/> </packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="_E"
name="Monopoly Game"/>

Output (OWL)
:Plays rdf:type owl:ObjectProperty;
rdfs:domain :Player;
rdfs:range :Monopoly Game;
:inv-Plays rdf:type owl:ObjectProperty;
rdfs:domain :Monopoly Game;
rdfs:range :Player;
:Plays owl:inverseOf :inv-Plays
```

4.1.5. Multiplicities

In the association relationship, multiplicities define how many instances of a class can be linked to how many instances of the other class. Association multiplicity can be defined in various ways such as range of values, an exact value, unlimited, and a set of distinct values. In ontology, the association multiplicity is represented by qualified cardinality in ***Range*** constructs. We specified UML association multiplicities in ontology through ***owl:minQualifiedCardinality*** and ***owl:maxQualifiedCardinalityc*** in ***owl:equivalentClass constraint***.

```
Rule (XSLT)
<xsl:if test="position()!=1">
<xsl:value-of select="key('Classid',./preceding-sibling::*
[1]/@type)/@name"/>
owl:equivalentClass [ rdf:type owl:Class ;
owl:intersectionOf ( [ rdf:type owl:Restriction
owl:onProperty :<xsl:value-of select="../@name"/>;
owl:onClass :<xsl:value-of select="key('Classid',@type)/@name"/>;
owl:minQualifiedCardinality "<xsl:choose>
<xsl:when test="not(./lowerValue)"> 1 </xsl:when>
<xsl:when test="./lowerValue/@value"> <xsl:value-of select=".
/lowerValue/@value"/> </xsl:when> <xsl:otherwise> 0 </xsl:otherwise>
</xsl:choose>"^^xsd:nonNegativeInteger] [ rdf:type owl:Restriction ;
owl:onProperty :<xsl:value-of select="../@name"/>;owl:onClass
:<xsl:value-of select="key('Classid',@type)/@name"/>;
owl:maxQualifiedCardinality "
<xsl:choose>
<xsl:when test="not(./upperValue)"> 1 </xsl:when>
<xsl:when test="./upperValue/@value"><xsl:value-of select="./
upperValue/@value"/> </xsl:when>
<xsl:otherwise> 0 </xsl:otherwise>
</xsl:choose>"^^xsd:nonNegativeInteger])]
</xsl:if>

Example (XMI)
<ownedEnd xmi:type="uml:Property" xmi:id="_bc1" association="Plays">
<upperValue xmi:type=" uml:LiteralInteger" xmi:id="_b1" value="1"/>
```

```
<lowerValue xmi:type="uml:LiteralInteger" xmi:id="_b2" value="1"/>
</ownedEnd>

Output (OWL)
rdfs:range [rdf:type owl:Class ;
owl:intersectionOf ([rdf:type owl:Restriction; owl:onProperty :Plays;
owl:onClass:Monopoly Game ;owl:minQualifiedCardinality
"1"^^xsd:nonNegativeInteger] [rdf:type owl:Restriction ;
owl:onProperty :Plays ; owl:onClass :Monopoly Game ;
owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger])] .
```

4.1.6. XOR Constraint

In the UML class model, a class can be associated with multiple associations with other class/classes. These associations can be mutually exclusive by XOR constraint, as shown in Figure 1, where ***Player*** and ***Unowned Property*** classes are linked through ***Buy*** and ***Auction*** associations. The XOR constraint can be applied to a single association when an association is asscoaited with more than one class, as shown in Figure 6, where the ***Account*** class is associated with the ***Company*** and ***Person*** classes through ***Belong*** association. In this case, the XOR constraint restricts the instance of a source class which can be linked only to the one target class instance.

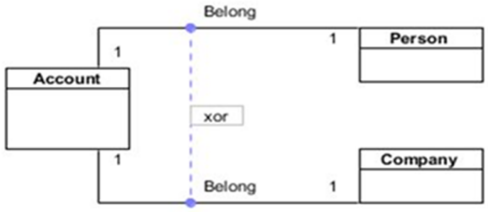

Figure 6. XOR Constraint on Single Association between Different Classes.

In the proposed tool, for the first case, two disjoint object properties are declared for XOR association constraint. Due to disjoint constraint, an instance of Player can be connected with an instance of Unowned Property through object property Buy or through object property Auction but not both of them.

```
Rule (XSLT)
<xsl:if test="@name='xor'">
<xsl:variable name="abc" select="tokenize(@constrainedElement,' ')"/>
<xsl:variable name= "key1" select="key('consid',$abc[1])/@association"/>
<xsl:variable name= "key2" select="key('consid',$abc[2])/@association"/>
<xsl:variable name="Ass1" select="key('Classid',$key1)/@name"/>
<xsl:variable name ="Ass2" select="key('Classid',$key2)/@name"/>
<xsl:variable name="Ass1End" select="key('Classid',$key1)/@memberEnd"/>
<xsl:variable name ="Ass2End" select="key('Classid',$key2)/@memberEnd"/>
<xsl:variable name="leftAssownedend1" select="mf:searchobject($Ass1End,2)"/>
<xsl:variable name="leftclassid1" select="key('consid',$leftAssownedend1)
/@type"/>
<xsl:variable name="rightAssownedend1" select="mf:searchobject($Ass1End,1)
"/>
<xsl:variable name="rightclassid1" select="key('consid',$rightAssownedend1)
/@type"/>
<xsl:variable name="rightAssownedend2" select="mf:searchobject($Ass2End,1)"/>
<xsl:variable name="rightclassid2" select="key('consid',$rightAssownedend2)
/@type"/>

Example (XMI)
<ownedRule xmi:type="uml:Constraint" xmi:id="_Q1" name="xor"
constrainedElement="_Q2 _Q1">
<specification xmi:type="uml:LiteralString" xmi:id="_Q2" value=""/>
</ownedRule>
<packagedElement xmi:type="uml:Class" xmi:id="_Q1" name="Player"/>
```

```
<packagedElement xmi:type="uml:Association" xmi:id="_Q3" name="Buy">
</packagedElement>
<packagedElement xmi:type="uml:Association" xmi:id="_Q6" name="Auction">
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="_Q11" name="Auction"/>

Output (OWL)
:Unowned Property rdf:type owl:Class.
:Players rdf:type owl:Class.
:Auction rdf:type owl:ObjectProperty ;
:Buy rdf:type owl:ObjectProperty ;
owl:propertyDisjointWith :Auction.
```

For the second case, the association is transformed into an object property with the constraint as shown below.

```
Rule (XSLT)
<xsl:when test ="$Ass1=$Ass2">
:<xsl:value-of select="$Ass1"/> rdf:type owl:ObjectProperty.
:<xsl:value-of select="key('Classid',$leftclassid1)/@name"/>
owl:equivalentClass [rdf:type owl:Class ;
owl:unionOf ([rdf:type owl:Class ;
owl:intersectionOf ([rdf:type owl:Class ;
owl:complementOf [rdf:type owl:Restriction ;
owl:onProperty :<xsl:value-of select="$Ass1"/> ;
owl:someValuesFrom :<xsl:value-of select="key('Classid',$rightclassid2)
/@name"/>;]]
[rdf:type owl:Restriction;owl:onProperty :<xsl:value-of select="$Ass1"/> ;
owl:someValuesFrom :<xsl:value-of select="key('Classid',$rightclassid1)
/@name"/>;])]
[ rdf:type owl:Class ;
owl:intersectionOf ( [ rdf:type owl:Class ;
owl:complementOf [ rdf:type owl:Restriction ;
owl:onProperty :<xsl:value-of select="$Ass1"/> ;
owl:someValuesFrom :<xsl:value-of select="key('Classid',$rightclassid1)
/@name"/>;]]
[ rdf:type owl:Restriction ;
owl:onProperty :<xsl:value-of select="$Ass1"/>;
owl:someValuesFrom :<xsl:value-of select="key('Classid',$rightclassid2)
/@name"/>;])])]. </xsl:when>

Example (XMI)
<packagedElement xmi:type="uml:Class" xmi:id="_L1" name="Account"/>
<packagedElement xmi:type="uml:Association" xmi:id="_L2" name="Belong">
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="_L10" name="Person"/>
<packagedElement xmi:type="uml:Class" xmi:id="_L11" name="Company">
</packagedElement>

Output (OWL)
:Account
owl:equivalentClass [rdf:type owl:Class ; owl:intersectionOf ([rdf:type owl:Class;
owl:intersectionOf ([rdf:type owl:Class ; owl:complementOf[ rdf:type owl:Restriction;
owl:onProperty :Belong; owl:someValuesFrom :Person]] [ rdf:type owl:Restriction;
owl:onProperty :Travel ; owl:someValuesFrom :CommercialVehicle])]
[rdf:type owl:Class ; owl:unionOf([rdf:type owl:Restriction;owl:onProperty :Belong;
owl:someValuesFrom :Vehicle] [ rdf:type owl:Restriction ; owl:onProperty :Travel;
owl:someValuesFrom :Company])])].
```

4.2. Reasoning Component

After transformation, the reasoning component performs model verification. The reasoning component initially passes the transformed ontology to the association satisfiability verifier sub-component, which uses an ontology-based algorithm to determine whether the association multiplicity constraints are finitely satisfiable, and the model has a finite number of elements. The ontology-based algorithm optimized the "detection graph technique" and improved the verification time through the search space reduction. It transformed the Class model into the ontology graph, as shown in Figure 2, and traverse the graph for finding the critical cycles. The tool does not transverse unimportant cycles (greater, balance cycles), which cause unnecessary delay in the verification process. Here is an abstract pseudo-code of the ontology-based verification of association constraints.

```
CheckCardinality()
for (Graph g :list)
findPath(g,g.getRange, g.getDomain);
CriticalCycle=false;
for (CardFeedBackObject f : fb)
if (f.criticalCycle)
CriticalCycle=true;
return CriticalCycle;
findPath(Graph g,String r, String s)
String p=g.getProperty();
Path.add(p);
weight=1.0;
while (true)
Graph g1=searchInverse(p);
weight *=g.getMax*(1.0/ g1.getMin);
g= searchInGraph(g.getRange,g.getProperty);
r=g.getRange;
p=g.getProperty;
Path.add(p);
if (s=r)
g1=searchInverse(p);
weight *=g.getMax*(1.0/ g1.getMin);
break;
```

If the association constraints are successfully verified, then the verification process moves toward the verification of XOR constraint and dependency realtionship. The XOR constraint verifier sub-component verifies the satisfiability of both types of XOR constraints through built-in ontology reasoner.

4.3. Feedback Component

Once the UML class model is verified successfully or fails in any test, the result of the verification is prepared by the feedback component. In the failure condition, a proper reason for the failure is generated by the feedback component, as shown in Figure 7.

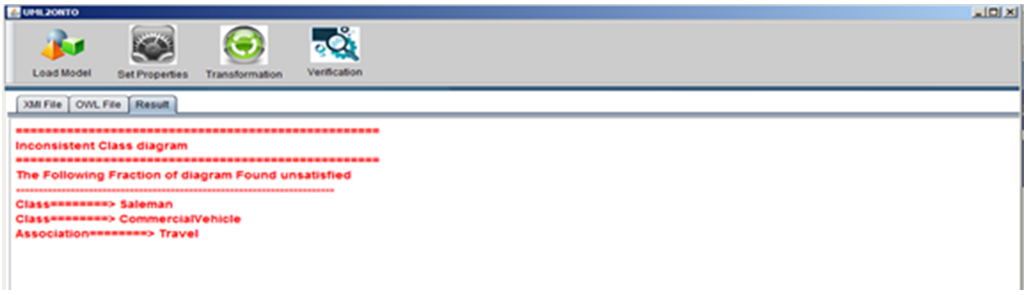

Figure 7. Verification Result Generated from UCLAONT.

5. Experimental Results

We implemented our approach as a Java prototype tool which has been already discussed in the previous section. For the subject of our study, we consider 11 different UML class models in which six models are used for association verification, 5 are used for XOR constraints. The Bellman-Ford algorithm [57] has been used to compare the proposed method because Bellman-ford is also a graph-based technique for detecting negative weight critical cycles. Many current verification techniques have employed the same with a little bit of adjustment for verification of satisfiability of association constraint. Furthermore, we also compared the proposed tool with UMLtoCSP and Alloy because they are widely used. To be justified, the proposed approach should provide an advantage over OCL based

approach that does not support direct verification of XOR constraint. We thus compared the performance of the proposed approach with UMLtoCSP and UML2Alloy, which support verification of the UML class model with OCL.

For the performance verification of the proposed tool, the experimental setup was made to run on a Core i7 machine with 4 GB of RAM. However, UMLtoCSP tool does not execute on 64-bit architecture due to this experimental setup was made to run on Core2Duo 1.34 GHz computer with two Gigabytes of RAM.

The implemented approach in UCLAONT is very prompt and can check the correctness of large and complex model within a few milliseconds, as shown in Table 1 and Figure 8. For example, a class model containing 100 classes and 100 associations can be verified in 0.68 s in Alloy [44] and 0.89 s in UMLtoCSP [15,43]. On the other side, UCLAONT takes only 0.003963 s. The presented tool offers two major advantages compared to other similar tools. First, it supports verification of large class model in a reasonable time, such as a class model containing 1000 classes and 1000 associations it takes only 0.090366 s. Secondly, it provides unbounded verification of XOR constraints and dependency relationships.

Table 1. Experimental Results.

Model	Classes	Associations	UMLTOCSP	Alloy	UMLtoONTO
Employee Department	3	5	0.032 s	0.032 s	0.00099 s
Travel Agency	12	12	0.51 s	0.51 s	0.00210 s
Pet Store	18	19	0.65 s	0.631 s	0.00328 s
Programmed 1	100	100	0.89 s	0.686 s	0.003963 s
Programmed 2	500	500	Timeout	3.890 s	0.04534 s
Programmed 3	1000	1000	Timeout	7.28 s	0.090366 s

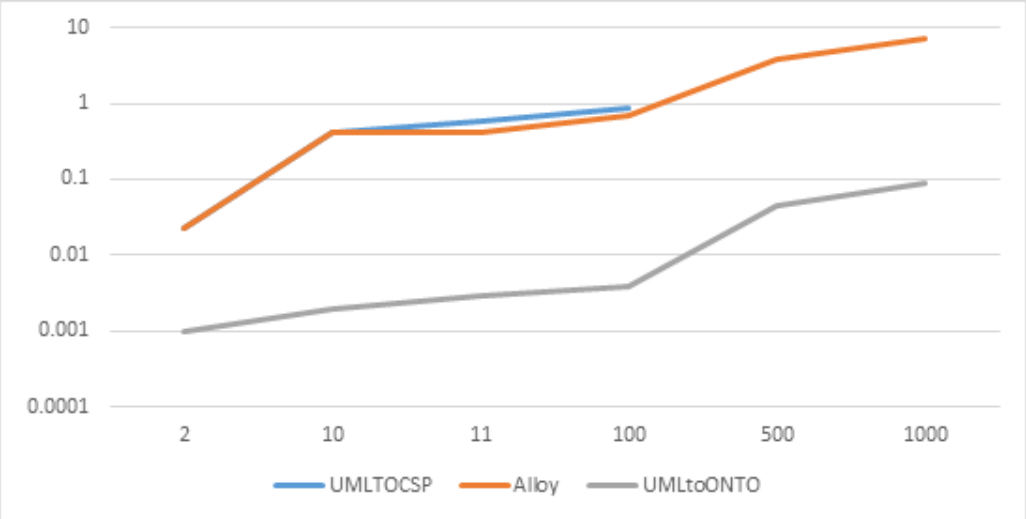

Figure 8. Verification Time of UML Class Models with UMLtoCSP, Alloy, and UCLAONT.

Table 2 presents the execution time to verify the XOR models. For the first model, that is, Project-Tasks, the proposed tool takes on average 0.035 s with three classes and 1 XOR association. In Order Management, which has XOR constraints of type 2 (XOR on multiple associations), the proposed tool takes on average 0.103 s with eight classes and three XOR association constraints. In the Restaurant model, which has XOR constraints of type 1 (XOR on single association), the proposed tool takes on average 0.100 s with eight classes and three XOR associations. the Meta-model of value properties attributes model, which has both types of XOR constraints takes on average 0.080 s. Finally, for checking the proposed tool's performance on a large model, an experiment is performed on Programmed 4, which

has 100 classes, 200 associations, 100 XOR constraints, and the proposed tool takes on average 0.570 s.

Table 2. Experimental Results.

Model Name	Classes	Associations	XOR	Case 1	Case 2	Verification Time
Project Tasks	4	3	1	✓	×	0.035
Order Management	6	8	3	×	✓	0.103
Meta-model of value Properties Attributes	4	10	3	✓	×	0.080
Restaurant	8	8	3	✓	✓	0.100
Programmed 4	100	200	100	×	✓	0.547

6. Conclusions and Future Work

Software Model verification tools are required in many software methodologies, including Model-Driven Engineering, Agile, rational unified process methodology, and so forth. Generally, a model verification tool checks the bugs' presence in the model. UML is an industry-standard and well-recognized modeling language among software practitioners. The UML class model is the essential UML model, and it is used in the analysis and design. In the existing literature, various correctness properties of the UML class model have been checked by vacuous tools. However, the two correctness properties: (1) consistency; and (2) satisfiability are the most frequently tested correctness properties. Consistency can be cover in satisfiability because consistency focuses on non-emptiness and finite satisfiability focuses on finiteness. This paper presents a research tool—UCLAONT—which verifies the UML class model. The tools can verify a complex class model which has thousands of classes and associations. Additionally, the tool supports verification of XOR constraints and dependency relationships, which did not support any existing tool. This tool uses an ontology-based graph algorithm to verify the class model and provides feedback to the user if it is unsatisfiable. As our future work, we plan to develop an Eclipse plugin along with other relationships such as aggregations.

Author Contributions: Conceptualization, A.R., A.H. and A.S.; Methodology, A.A. and A.S.; Software, A.H.; Validation, M.S.A.R., M.H. and K.R.; Formal Analysis, A.H.; Investigation, A.R. and M.S.A.R.; Data Curation, M.H. and K.R.; Writing—Original Draft Preparation, A.R., A.H. and A.S.; Writing—Review & Editing, A.A. and M.S.A.R.; Visualization, M.H.; Supervision, K.R.; Funding Acquisition, A.A. All authors have read and agreed to the published version of the manuscript.

Funding: The authors would like to express their gratitude to the ministry of education and the deanship of scientific research Najran University Kingdom of Saudi Arabia for their financial and technical support under code number NU/-/SERC/10/554.

Data Availability Statement: All the data is available within the article and relevant software code for this research work are stored in GitHub and can be downloaded from this link https://github.com/hafeez1978/UMLtoON, accessed on 30 August 2021.

Conflicts of Interest: The authors declare that they have no conflict of interest to report regarding the present study.

References

1. Eriksson, H.E.; Penker, M. *Business Modeling with UML*; Citeseer: New York, NY, USA, 2000 ; pp. 1–12.
2. Rhazali, Y.; Moulay Youssef, H.; Mouloudi, A. A Model transformation from computing independent model to platform independent model in model driven architecture. *J. Ubiquitous Syst. Pervasive Netw.* **2017**, *8*, 19–26. [CrossRef]
3. Biehl, M. Literature study on model transformations. *R. Inst. Technol. Tech. Rep. ISRN/KTH/MMK* **2010**, *291*, 1–28.
4. Mens, T.; Van Gorp, P. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.* **2006**, *152*, 125–142. [CrossRef]
5. Meedeniya, D.A. Correct Model-to-Model Transformation for Formal Verification. Ph.D. Thesis, University of St Andrews, St Andrews, UK, 2013.
6. Singh, M.; Sharma, A.K.; Saxena, R. An UML+ Z framework for validating and verifying the Static aspect of Safety Critical System. *Procedia Comput. Sci.* **2016**, *85*, 352–361. [CrossRef]

7. Dwyer, M.B.; Hatcliff, J.; Robby, R.; Pasareanu, C.S.; Visser, W. Formal software analysis emerging trends in software model checking. In Proceedings of the 2007 Future of Software Engineering, Minneapolis, MN, USA, 23–25 May 2007 ; pp. 120–136.
8. Filax, M.; Gonschorek, T.; Ortmeier, F. Correct formalization of requirement specifications: A v-model for building formal models. In *International Conference on Reliability, Safety, and Security of Railway Systems*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 106–122.
9. Bowen, J.P. *Formal Specification and Documentation Using Z: A Case Study Approach*; International Thomson Computer Press: London, UK, 1996; Volume 66.
10. Beato, M.E.; Barrio-Solórzano, M.; Cuesta, C.E.; de la Fuente, P. UML automatic verification tool with formal methods. *Electron. Notes Theor. Comput. Sci.* **2005**, *127*, 3–16. [CrossRef]
11. Kobryn, C. UML 2001: A standardization odyssey. *Commun. ACM* **1999**, *42*, 29–37. [CrossRef]
12. Dobing, B.; Parsons, J. How UML is used. *Commun. ACM* **2006**, *49*, 109–113. [CrossRef]
13. Shaikh, A.; Wiil, U.K. A feedback technique for unsatisfiable UML/OCL class diagrams. *Softw. Pract. Exp.* **2014**, *44*, 1379–1393. [CrossRef]
14. Shaikh, A.; Wiil, U.K.; Memon, N. Evaluation of tools and slicing techniques for efficient verification of UML/OCL class diagrams. *Adv. Softw. Eng.* **2011**, *2011*, 370198. [CrossRef]
15. Shaikh, A.; Wiil, U.K. UMLtoCSP (UOST): A tool for efficient verification of UML/OCL class diagrams through model slicing. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Cary, NC, USA, 11–16 November 2012; ACM: New York, NY, USA, 2012; p. 37.
16. Rumbaugh, J.; Jacobson, I.; Booch, G. *The Unified Modeling Language Reference Manual*; Pearson Higher Education: Hoboken, NJ, USA, 2004.
17. Batory, D.; Azanza, M. Teaching model-driven engineering from a relational database perspective. *Softw. Syst. Model.* **2017**, *16*, 443–467. [CrossRef]
18. Hilken, F.; Gogolla, M. User Assistance Characteristics of the USE Model Checking Tool. *arXiv* **2017**, arXiv:1701.08471.
19. Cabot, J.; Clarisó, R. UML/OCL verification in practice. In Proceedings of the ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering, Toulouse, France, 28 September–3 October 2008; ACM: New York, NY, USA, 2008; pp. 31–35.
20. Przigoda, N.; Soeken, M.; Wille, R.; Drechsler, R. Verifying the structure and behavior in UML/OCL models using satisfiability solvers. *IET Cyber-Phys. Syst. Theory Appl.* **2016**, *1*, 49–59. [CrossRef]
21. Shaikh, A.; Clarisó, R.; Wiil, U.K.; Memon, N. Verification-driven slicing of UML/OCL models. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, 20–24 September 2010; ACM: New York, NY, USA, 2010; pp. 185–194.
22. Cabot, J.; Claris, R.; Riera, D. Verification of UML/OCL class diagrams using constraint programming. In Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, Lillehammer, Norway, 9–11 April 2008; pp. 73–80.
23. Object Management Group. OMG Unified Modeling Language TM (OMG UML) Superstructure v. 2.3. *InformatikSpektrum* **2010**, *21*, 758.
24. Pandey, R. Object constraint language (OCL) past, present and future. *ACM Sigsoft Softw. Eng. Notes* **2011**, *36*, 1–4. [CrossRef]
25. Balaban, M.; Maraee, A. Finite satisfiability of UML class diagrams with constrained class hierarchy. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2013**, *22*, 24. [CrossRef]
26. Shaikh, A.; Hafeez, A.; Elmagzoub, M.; Alghamdi, A.; Siddique, A.; Shahzad, B. Ontology-Based Verification of UML Class Model XOR Constraint and Dependency Relationship Constraints. *Intell. Autom. Soft Comput.* **2021**, *27*, 565–579. [CrossRef]
27. Khan, A.H.; Musavi, S.H.A.; Rehman, A.U.; Shaikh, A. Ontology-Based Finite Satisfiability of UML Class Model. *IEEE Access* **2018**, *6*, 3040–3050. [CrossRef]
28. Hafeez, A.; Abbas, S.; Rehman, A. Ontology-Based Transformation and Verification of UML Class Model. *Int. Arab. J. Inf. Technol.* **2020**, *7*, 758–768. [CrossRef]
29. Shaikh, A.; Khan, A.H.; Wagan, A.A.; Alrizq, M.; Alghamdi, A.; Reshan, M.S.A. More Than Two Decades of Research on Verification of UML Class Models: A Systematic Literature Review. *IEEE Access* **2021**, *9*, 142461–142474. [CrossRef]
30. Truong, N.T.; Souquières, J. An approach for the verification of UML models using B. In Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, Brno, Czech Republic, 27 May 2004; pp. 195–202.
31. He, H.; Wang, Z.; Dong, Q.; Zhang, W.; Zhu, W. Ontology-based semantic verification for uml behavioral models. *Int. J. Softw. Eng. Knowl. Eng.* **2013**, *23*, 117–145. [CrossRef]
32. Berardi, D.; Calvanese, D.; De Giacomo, G. Reasoning on UML class diagrams. *Artif. Intell.* **2005**, *168*, 70–118. [CrossRef]
33. France, R.; Evans, A.; Lano, K.; Rumpe, B. The UML as a formal modeling notation. *Comput. Stand. Interfaces* **1998**, *19*, 325–334. [CrossRef]
34. Maoz, S.; Ringert, J.O.; Rumpe, B. CD2Alloy: Class diagrams analysis using Alloy revisited. In Proceedings of the International Conference on Model Driven Engineering Languages and Systems, Wellington, New Zealand, 16–21 October 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 592–607.
35. Anastasakis, K.; Bordbar, B.; Georg, G.; Ray, I. On challenges of model transformation from UML to Alloy. *Softw. Syst. Model.* **2010**, *9*, 69. [CrossRef]

36. Artale, A.; Calvanese, D.; Ibáñez-García, A. Full satisfiability of UML class diagrams. In Proceedings of the International Conference on Conceptual Modeling, Vancouver, BC, Canada, 1–4 November 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 317–331.
37. Maraee, A.; Balaban, M. Efficient recognition of finite satisfiability in UML class diagrams: Strengthening by propagation of disjoint constraints. In Proceedings of the 2009 International Conference on Model-Based Systems Engineering, Herzeliya and Haifa, Israel, 2–5 March 2009; pp. 1–8.
38. Malgouyres, H.; Motet, G. A UML model consistency verification approach based on meta-modeling formalization. In Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, 23–27 April 2006; ACM: New York, NY, USA, 2006; pp. 1804–1809.
39. Cadoli, M.; Calvanese, D.; De Giacomo, G.; Mancini, T. Finite satisfiability of UML class diagrams by Constraint Programming. *CSP Tech. Immed. Appl. (CSPIA)* **2004**, *2*, 2–16.
40. Cabot, J.; Teniente, E. Incremental integrity checking of UML/OCL conceptual schemas. *J. Syst. Softw.* **2009**, *82*, 1459–1478. [CrossRef]
41. Ledang, H. Automatic translation from UML specifications to B. In Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001), San Diego, CA, USA, 26–29 November 2001; p. 436.
42. Marcano, R.; Levy, N. Using B formal specifications for analysis and verification of UML/OCL models. In *Workshop on Consistency Problems in UML-Based Software Development, Proceedings of the 5th International Conference on the Unified Modeling Language, Dresden, Germany, 30 September–4 October 2002*; Citeseer: New York, NY, USA, 2002; pp. 91–105.
43. Cabot, J.; Clarisó, R.; Riera, D. 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, Atlanta, GA, USA, 5–9 November 2007; ACM: New York, NY, USA, 2007; pp. 547–548.
44. Bordbar, B.; Anastasakis, K. UML2ALLOY: A Tool for Lightweight Modelling of Discrete Event Systems. In Proceedings of the IADIS International Conference on Applied Computing, Algarve, Portugal, 22–25 February 2005; pp. 209–216.
45. Nguyen, T.H.; Grundy, J.C.; Almorsy, M. Ontology-based automated support for goal–use case model analysis. *Softw. Qual. J.* **2016**, *24*, 635–673. [CrossRef]
46. Corea, C.; Delfmann, P. Detecting Compliance with Business Rules in Ontology-Based Process Modeling. 2017. Available online: https://aisel.aisnet.org/wi2017/track03/paper/6/ (accessed on 30 August 2021).
47. Fellmann, M.; Hogrebe, F.; Thomas, O.; Nüttgens, M. An ontology-driven approach to support semantic verification in business process modeling. In Proceedings of the Modellierung Betrieblicher Informationssysteme (MobIS 2010), Modellgestütztes Management, Dresden, Germany, 15–17 September 2010.
48. Sun, J.; Wang, H.H.; Hu, T. Design Software Architecture Models using Ontology. In Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering, Miami, FL, USA, 7–9 July 2011; pp. 191–196.
49. Kezadri, M.; Pantel, M. *First Steps Toward a Verification and Validation Ontology*; Embedded Real Time Software and Systems (ERTS2012): Toulouse, France, 2012; pp. 440–444.
50. Xu, W.; Dilo, A.; Zlatanova, S.; van Oosterom, P. *Modelling Emergency Response Processes: Comparative Study on OWL and UML*; Information Systems for Crisis Response and Management, Harbin Engineering University: Harbin, China, 2008; pp. 493–504.
51. Bahaj, M.; Bakkas, J. Automatic conversion method of class diagrams to ontologies maintaining their semantic features. *Int. J. Soft Comput. Eng. (IJSCE)* **2013**, *2*, 65.
52. Belghiat, A.; Bourahla, M. From UML Class Diagrams to OWL Ontologies: A Graph Transformation Based Approach. In Proceedings of the 4th International Conference on Web and Information Technologies ICWIT 2012, Sidi Bel Abbes, Algeria, 29–30 April 2012; pp. 330–335.
53. Parreiras, F.S.; Staab, S. Using ontologies with UML class-based modeling: The TwoUse approach. *Data Knowl. Eng.* **2010**, *69*, 1194–1207. [CrossRef]
54. Clarisó, R.; González, C.A.; Cabot, J. Incremental Verification of UML/OCL Models. *J. Object Technol.* **2020**, *19*, 3. [CrossRef]
55. Abbas, M.; Ben-Yelles, C.B.; Rioboo, R. Formalizing UML/OCL structural features with FoCaLiZe. *Soft Comput.* **2020**, *24*, 4149–4164. [CrossRef]
56. Pérez, B.; Porres, I. Reasoning about UML/OCL class diagrams using constraint logic programming and formula. *Inf. Syst.* **2019**, *81*, 152–177. [CrossRef]
57. Goldberg, A.; Radzik, T. *A Heuristic Improvement of the Bellman-Ford Algorithm*; Technical Report; Computer Science Department, Stanford University: Stanford, CA, USA, 1993.