Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2011, Article ID 370198, 18 pages
doi:10.1155/2011/370198

Research Article

Evaluation of Tools and Slicing Techniques for Efficient Verification of UML/OCL Class Diagrams

Asadullah Shaikh,[1,2] Uffe Kock Wiil,[1] and Nasrullah Memon[1,3]

[1] *The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, 5230 Odense, Denmark*
[2] *Universitat Oberta de Catalunya, Barcelona 08018, Spain*
[3] *Mehran University of Engineering & Technology, Jamshoro 76062, Pakistan*

Correspondence should be addressed to Asadullah Shaikh, shaikhasad@hotmail.com

Received 31 December 2010; Revised 9 May 2011; Accepted 28 June 2011

Academic Editor: Andrea De Lucia

UML/OCL class diagrams provide high-level descriptions of software systems. Currently, UML/OCL class diagrams are highly used for code generation through several transformations in order to save time and effort of software developers. Therefore, verification of these class diagrams is essential in order to generate accurate transformations. Verification of UML/OCL class diagrams is a quite challenging task when the input is large (i.e., a complex UML/OCL class diagram). In this paper, we present (1) a benchmark for UML/OCL verification and validation tools, (2) an evaluation and analysis of tools available for verification and validation of UML/OCL class diagrams including the range of UML support for each tool, (3) the problems with efficiency of the verification process for UML/OCL class diagrams, and (4) solution for efficient verification of complex class diagrams.

1. Introduction

UML/OCL models are designed in order to provide a high-level description of a software system which can be used as a piece of documentation or as an intermediate step in the software development process. In the context of model-driven development (MDD) and model-driven architecture (MDA), a correct specification is required because the entire technology is based on model transformation. Therefore, if the original model is wrong, this clearly causes a failure of the final software system. Regrettably, verification of a software product is a complex and time-consuming task [1] and that also applies to the analysis of the software models. With increasing model size and complexity, the need for efficient verification methods able to cope with the growing difficulties is ever present, and the importance of UML models has increased significantly [2].

There are formal verification tools for automatically checking correctness properties on models [3–6], but the lack of scalability is usually a drawback. We have considered the static structure diagram that describes the structure of a system, modeled as a UML class diagram. Complex integrity constraints will be expressed in OCL. In this context, the fundamental correctness property of a model is *satisfiability*.

We focus our discussion on the verification of a specific property: satisfiability—"is it possible to create objects without violating any constraints.?" The property is relevant in the sense that many interesting properties; for example, redundancy of an integrity constraint can be expressed in terms of satisfiability [7]. Two different notions of satisfiability can be checked, either weak satisfiability or strong satisfiability. A class diagram is weakly satisfiable if it is possible to create a legal instance/object of a class diagram which is non mpty; that is, it contains at least one object from some class. Alternatively, strong satisfiability is a more restrictive condition requiring that the legal instance has at least one object from each class and a link from each association [4].

Reasoning on UML class diagrams without OCL integrity constraints is an EXPTIME-complete problem [8]. The UML class diagram analysis is a complex problem. However, the addition of OCL constraint makes the problem undecidable in general. In order to avoid this undecidability, the automatic verification procedure is required. Furthermore,

the addition of unrestricted (Some approaches restrict the set of supported OCL constructs, e.g., to make the verification decidable. In this paper, we consider general OCL constraints with no limitations on their expressivity.) OCL constraints makes the problem undecidable. The unrestricted form of OCL constraints is not limited in expressiveness to equality, size, and attribute operations.

Current solutions for checking satisfiability employ formalisms such as description logics [9], higher-order logics [10], database deduction systems [11], linear programming [12], SAT [13], or constraint satisfaction problems [4, 8]. However, all the approaches which support general OCL constraints share a common drawback, a high worst-case computational complexity. Their execution time may depend exponentially on the size of the model, understanding size as the number of classes/attributes/associations in the model, and/or the number of OCL constraints [14–16].

Hitherto, the existing tools are only capable of verifying model properties; such tools verify the model properties much quicker when the input is a small UML/OCL class diagram. In case of a large UML/OCL class diagram as input, these tools become so inefficient that verification becomes impractical. Therefore, there is a need for efficient verification tool or technique to cope with this complexity problem. As the complexity of a model can be exponential in terms of model size (i.e., the number of classes, associations, and inheritance hierarchies), reducing the size of a model can cause a drastic speedup in the verification process. One possible approach is *slicing*, which is partitioning the class diagram and OCL constraints into smaller fragments according to certain criteria. This partition should preserve the property under verification in the sense that it should be possible to assess the property in the original model from the analysis of the partitions. A careful definition of the partition process is needed to ensure this.

In this paper, we propose a survey and a benchmark based on analysis of current formal verification tools. The survey is based on

(i) identification of verification and validation tools,

(ii) evaluation of current verification and validation tools,

(iii) the UML support for each verification and validation tool.

Furthermore, we have identified some issues related to the efficiency of verification of UML/OCL class diagrams. Therefore, our focus is to address those issues along with an appropriate solution. The solution is named as UML/OCL slicing technique (UOST). The technique includes a set of heuristics that are used to partition a model when determining its satisfiability. That is, given a model "m", the technique partitions m into $m_1, m_2, m_3, \ldots, m_n$ submodels, where m is satisfiable if all $m_1, m_2, m_3, \ldots, m_n$ submodels are satisfiable. We provide an experimental evaluation of this technique using a verification tool for UML to CSP which is called UMLtoCSP [4] and Alloy [6]. We examine both small and large UML/OCL class diagrams with 2, 15, 17, 50, 100, 500, and 1000 classes and several OCL invariants to measure the efficiency of the verification process through our proposed UOST. It also provides extensive results achieved by adding the slicing technique to an external tool (Alloy). The primary reason for showing the results in both Alloy and UMLtoCSP is to demonstrate that the developed slicing technique is neither tool dependent nor formalism dependent. It can be applied into any formal verification tool for UML/OCL models.

The remainder of the paper is structured as follows. Section 2 provides a description and evaluation of selected UML/OCL tools. It also presents an efficiency analysis based on two of the tools. In Section 3, a solution to the identified efficiency problem is presented while Section 4 is comprised of nondisjoint solution. Section 5 presents the experimental results based on the proposed solution. In Section 6, related work is presented. Finally, Section 7 provides the conclusions and identifies directions for future work.

2. Evaluation of Existing UML/OCL Tools

This section presents a description of UML/OCL tools and methods that are widely used for the purpose of verification and validation of UML class diagrams. After evaluation of existing UML/OCL tools, we have discovered certain limitations in most of the verification and validation tools.

2.1. Analysis of UML/OCL Tools. The analysis is based upon a close view of each UML/OCL tool including the range of support for UML by each tool. The purpose of UML/OCL tools is to check the model properties that need to be verified. We have analyzed six UML/OCL tools in all. The first four subsections 2.1.1–2.1.4 discuss the verification tools: (1) HOL-OCL, (2) UMLtoCSP, (3) Alloy, (4) UML2Alloy. The last two subsections 2.1.5 and 2.1.6 discuss the validation tools: (5) USE and (6) MOVA. Each tool focuses on a different formalism and verification procedure. Therefore, the verification time is entirely based on a specific formalism. There are several environments that provide verification capability for UML/OCL class diagrams. Each environment has several advantages and disadvantages summarized in this section.

2.1.1. HOL-OCL. Higher-order logic and object constraint language (HOL-OCL) is a theorem-proving environment that is incorporated in the model driven engineering (MDE) framework [17]. It is based on the UML [18] and OCL [19] specification annotated into Isabelle/HOL [10]. In principle the HOL-OCL is based in su4sml [17] and Isabelle/HOL, which is UML/OCL repository. Also the interface consists of SML of Isabelle/HOL. Through HOL-OCL, satisfiability of class invariants can be proved.

The purpose of HOL-OCL is to support the subsets of "UML core" related to UML class diagrams. The limitations of HOL-OCL are that it does not support qualifiers for association ends, enumeration and association classes, but it supports the association relations and represents those relations by association's ends.

HOL-OCL supports only the standard OCL data types, that is, Integer, Real, String, and Boolean, but these data types

are considered different than UML standard data types, that is, Integer, UnlimitedInteger, String, and Enumeration.

HOL-OCL is a theorem prover that needs user assistance because it does not support automations. This is a limitation of HOL-OCL. Moreover there are some data types that cannot be modeled explicitly into HOL-OCL-like OclVoid, OclModelElementType, and OclType [3].

2.1.2. UMLtoCSP. UMLtoCSP is a tool for formal verification of UML/OCL class diagrams [4], which is based on constraint solver as the verification engine. It is designed as a verification tool, which is limited to take ArgoUML-generated XMI [20] class diagrams as an input along with OCL text files and generate the results automatically in the form of "Yes" and "No." UMLtoCSP checks the correctness of model properties, such as strong and weak satisfiability of a model or the lack of redundant constraints. This tool translates the model into a constraint satisfaction problem (CSP), and then it relies on the constraint solver ECLiPSe [21] to verify if the solution exists in CSP or not. The whole process is fully automated and does not require any manual interaction during the verification process.

The development of the tool is in Java along with ECLiPSe constraint libraries with combined libraries of Dresden toolkit [22] that are used for parsing in UMLtoCSP. MDR is used for the import/export of XMI.

In terms of UML support, this tool is limited to class diagrams. It accepts the model with associations and generalization but it lacks dependencies, aggregations, and stereotypes. In terms of OCL it only supports writing invariants with pre- and postconditions, while query operations are not supported at all.

2.1.3. Alloy. Alloy is a language [6] which is used to describe structural properties. It uses first-order logic (FOL) for modeling design and is regarded as a lightweight design tool which permits devise designs properly and checks its correctness using the alloy analyzer. The designer formulates a design in the Alloy language, which is FOL dependent on relations, and checks the property correctness using alloy analyzer. The analyzer translates the alloy formula into Boolean Formula in conjunctive normal form (CNF) and resolute it with SAT solver.

Alloy does not take a UML class diagram as input. The model in Alloy must be written textually using the Alloy graphical user interface. Alloy syntax is compatible with UML; the assessment is quite complicated because the model does not contain any textual syntax; therefore, every model requires a set of constraints in order to perform verification. The prototype of Alloy is made in Java solver "SAT4J" in order to run on every platform.

2.1.4. UML2Alloy. UML2Alloy is research tool which is used to verify UML using Alloy in the technology of model-driven development (MDD) [23]. UML2Alloy is playing an important role to create a bridge between UML and Alloy. Users can take advantage of UML2Alloy by applying the benefits of the Alloy analyzer to UML class diagrams.

The tool takes an ArgoUML-generated XMI file in order to transform the UML model into Alloy code. It transforms the input into assertions, simulations, or invariants. The tool translates the classes into signatures and subclasses in subsignatures. If the class is abstract, then it is considered as an abstract signature. It translates the class attributes into signature fields.

Furthermore, as far as the OCL is concerned, it does not support shorthand OCL notations. The class invariants are normally converted to Alloy facts. Invariants can also be translated into assertions. Initially invariants are translated to a predicate and then a fact statement.

2.1.5. USE. UML-based specification environment (USE) is a tool used for the validation of UML and OCL Models [24]. The formalism behind USE is A snapshot sequence language (ASSL). The construction of an ASSL script is manual, as the user needs to define the order in which objects will be created. At this point, the reader may not see that ASSL scripts normally describe sets of snapshots (system states); that is, ASSL scripts determine search spaces which have to be searched by the ASSL generator. This feature is enabled through the ASSL "try" statements which try different system state constellations out, inducing backtracking if the considered constellation does not lead to a valid snapshot. As a consequence, the order of object creations does not need to be manually determined.

The validation in USE is based on manual insertion; it allows the analysis of specification in order to validate the UML class diagram and its OCL constraints to avoid the defects in early stages of the development process. USE has a wide range of UML support; it fully provides validation syntax of class diagrams, sequence diagrams, and activity diagrams, while none of the above tools supports that much. In terms of OCL, it fully supports writing invariants using pre- and postconditions.

2.1.6. MOVA. MOVA is a tool [25], which is based on UML modeling. It is designed as a validation tool, which allows writing OCL constraints for UML class diagrams, drawing the instances of class diagram and validating OCL constraints over the instances of a class diagram. The tool has a user interface which permits the user to draw UML classes and object diagrams. It allows writing and verifying invariants and writing and assessing queries.

MOVA was developed initially in Java for the ITP/OCL tool [26]. It is a validation tool along with text input UML modeling. The tool provides a set of commands for building classes and object diagrams. It is written in Maude [27] which is a rewriting-based programming language that implements membership equation logic. The process of the MOVA graphical user interface is altered in ITP/OCL text-input commands and executed in Maude.

Mova has limited support for OCL and UML. In terms of UML, it supports drawing of classes, objects, associations, and generalizations while in terms of OCL it allows writing invariants, instances of UML classes without any support of pre- and postconditions at all.

Table 1: Comparison of tools based on formalism, verification, and translation.

Tool	Formalism	Verification	Translation	Limitations
HOL-OCL	Higher-Order logic	User-assisted	Automatic	Undecidability
UMLtoCSP	CSP	Automatic	Automatic	Accepts ArgoUML class diagram only and bounded verification
Alloy	Relational Logics	Automatic	Manual	No operation support involving integers
UML2Alloy	Alloy analyzer simulation	Automatic	Automatic	Accepts ArgoUML class diagram only
USE	ASSL	Automatic	Manual	Validation only
Mova	Maude	Automatic	Manual	Validation only

Table 2: Comparison of UML support for each verification and validation tool.

Relationship name	HOL-OCL	UMLtoCSP	Alloy	UML2Alloy	USE	MOVA
Abstraction	No	No	No	No	No	No
Aggregation	Yes	No	No	Yes	Yes	No
Association	Yes	Yes	Yes	Yes	Yes	Yes
Association classes	Yes	Yes	Yes	Yes	Yes	No
Binding	No	No	No	No	No	No
Composition association	Yes	No	No	No	No	No
Dependency	No	No	No	No	No	No
Generalization	Yes	Yes	Yes	Yes	Yes	Yes
Interface realization	No	No	No	No	No	No
Instantiation	No	No	No	No	No	No
Realization	No	No	No	No	No	No
Usage	No	No	No	No	No	No
N-ary	No	No	No	No	Yes	No

2.2. Comparison and Support of UML/OCL Tools. This section is based on the comparison and benchmark of UML/OCL tools discussed in Section 2.1. Generally, the comparison of UML/OCL tools is a challenging task whenever different formalisms are involved. Each verification method has different specification, and, therefore, the process of verification is also dissimilar. Currently, several challenges are being faced by researchers/developers just because of not having coordination among these tools. Table 1 shows a simple comparison between tools, used formalisms, verification, translation, and limitations.

We have developed several UML/OCL class diagrams which were verified by various verification and validation tools such as HOL-OCL, UMLtoCSP, Alloy, UML2Alloy, USE, and Mova. To the best of our knowledge, these tools are widely used for verification and validation purpose. Therefore, for each UML/OCL tool, we have provided a different range of UML supporting features. With the help of this support, a researcher can readily identify the tool support pertinent to the supported examples, while ignoring those instances where this support is not recorded. This would result in considerable saving of time and effort for the designers. Table 2 briefly compares the UML class relationship for different verification and validation tools, and Table 3 briefly compares the support of different stereotypes.

A class diagram defines different relationships between different objects of classes and all the possible relationships that appear in a class diagram such as associations, generalizations, and dependencies. In UML a stereotype is one way to extend the core semantics of the modeling language to express new things.

2.3. Efficiency Analysis of UML/OCL Tools. For the sake of brevity and without loss of generality, in this section we have examined the verification time in UMLtoCSP and Alloy. We have chosen UMLtoCSP and Alloy for our experimental results because these tools are widely used for verification, while Mova and USE are validation tools. UML2Alloy supports transformation of UML/OCL class diagram into Alloy specification; therefore, the experimental results are same as in Alloy. HOL-OCL is an interactive proof environment for OCL which we will consider for our future experiments. For each example, we have used the following parameters for the experiment: each class may have at most 4 instances, associations may have at most 1010 links, and attributes may range from 0 to 1022 for UMLtoCSP. Table 4 describes the set of benchmarks used for our comparison: the number of classes, associations, invariants, and attributes. The column "Verification Time" highlights the time taken by the tool to generate valid instances of a class diagram. The benchmarks

Table 3: Comparison of UML stereotypes by each tool.

Stereotype	HOL-OCL	MOVA	UMLtoCSP	Alloy	UML2Alloy	USE
Auxiliary	No	No	No	Yes	No	No
Enumeration	Yes	Yes	No	Yes	Yes	Yes
Type	No	No	No	No	No	No

Table 4: Description of UML/OCL benchmarks and verification time (UMLtoCSP).

Example	Classes	Associations	Attributes	Invariants	Verification Time
Paper-Researcher	2	2	6	1	0.04 s
Coach	15	12	2	2	5008.76 s
DBLP	17	27	38	26	Time-out
Tracking System	50	60	72	5	3605.35 s
Script 1	100	110	122	2	Time-out
Script 2	500	510	522	5	Time-out
Script 3	1000	1010	1022	5	Time-out

"Script" were programmatically generated in order to test large input class diagrams. The models "Paper-Researcher" and "Coach" serve as worst-case scenarios (models with many interdependent constraints) for verification time. Each experiment is conducted using a Intel Core 2 Duo Processor 2.1 Ghz with 2 Gb of RAM. All times are measured in seconds, and a time-out limit has been set at 2 hours (7200 seconds).

Similarly, we have programmed the Digital Bibliography and Library Project (DBLP) structural schema in the Alloy specification. The schema of the DBLP system is modeled as a UML class diagram [28]. The class diagram has 17 classes and 26 integrity constraints. Table 5 summarizes the verification time (TVT) obtained using the Alloy analyzer where Column (TT) is the translation time, Column (ST) is the solving time, and the summation of the TT and ST is the total verification time (TVT).

The above experiments show that verification is a time-consuming process for complex UML/OCL class diagrams.

2.4. Findings. Table 6 shows the findings related to the efficiency of the verification process. The verification time is largely depending on the UML/OCL model size and its complexity. Based on our analysis, we believe that this is a common problem for all such UML/OCL verification tools and methods and that these tools are unable to verify complex UML/OCL class diagrams. Therefore, an efficient method/technique that addresses is required the problem by reducing the verification time.

Table 5: Description of experimental results (Alloy).

Scope	TT	ST	TT + ST = TVT
2	0.125 s	0.047 s	0.172 s
3	0.187 s	0.078 s	0.265 s
4	0.281 s	0.172 s	0.453 s
5	0.473 s	0.190 s	0.663 s
6	0.671 s	0.344 s	1.015 s
7	0.969 s	0.484 s	1.453 s
⋮	⋮	⋮	⋮
1000	Time-out	Time-out	Time-out

TT: Translation Time. ST: Solving Time. TVT: Total Verification Time.

Table 6: Efficiency analysis for UMLtoCSP and Alloy.

UMLtoCSP		Alloy	
Classes	Efficiency	Scope	Efficiency
2	0.04 s	2	0.172 s
15	5008.76 s	3	0.265 s
50	3605.35 s	4	0.453 s
100	Time-out	5	0.663 s
500	Time-out	6	1.015 s
1000	Time-out	7	1.453 s
⋮	⋮	⋮	⋮
1050	Time-out	1000	Time-out

3. UML/OCL Model Slicing

The input of our method is a UML class diagram annotated with OCL invariants. Figure 1 introduces a class diagram that will be used as an example; the diagram models the information system of a bus company. Several integrity constraints are defined as OCL invariants.

Two different notions of satisfiability will be considered for verification: *strong* satisfiability and *weak* satisfiability. A class diagram is weakly satisfiable if it is possible to create at least one instance of at least one class out of all classes in the class diagram. Alternatively, in the case of strong satisfiability, it is an obligation that at least one object of all classes must be instantiated [4]. For example, it is possible that objects of

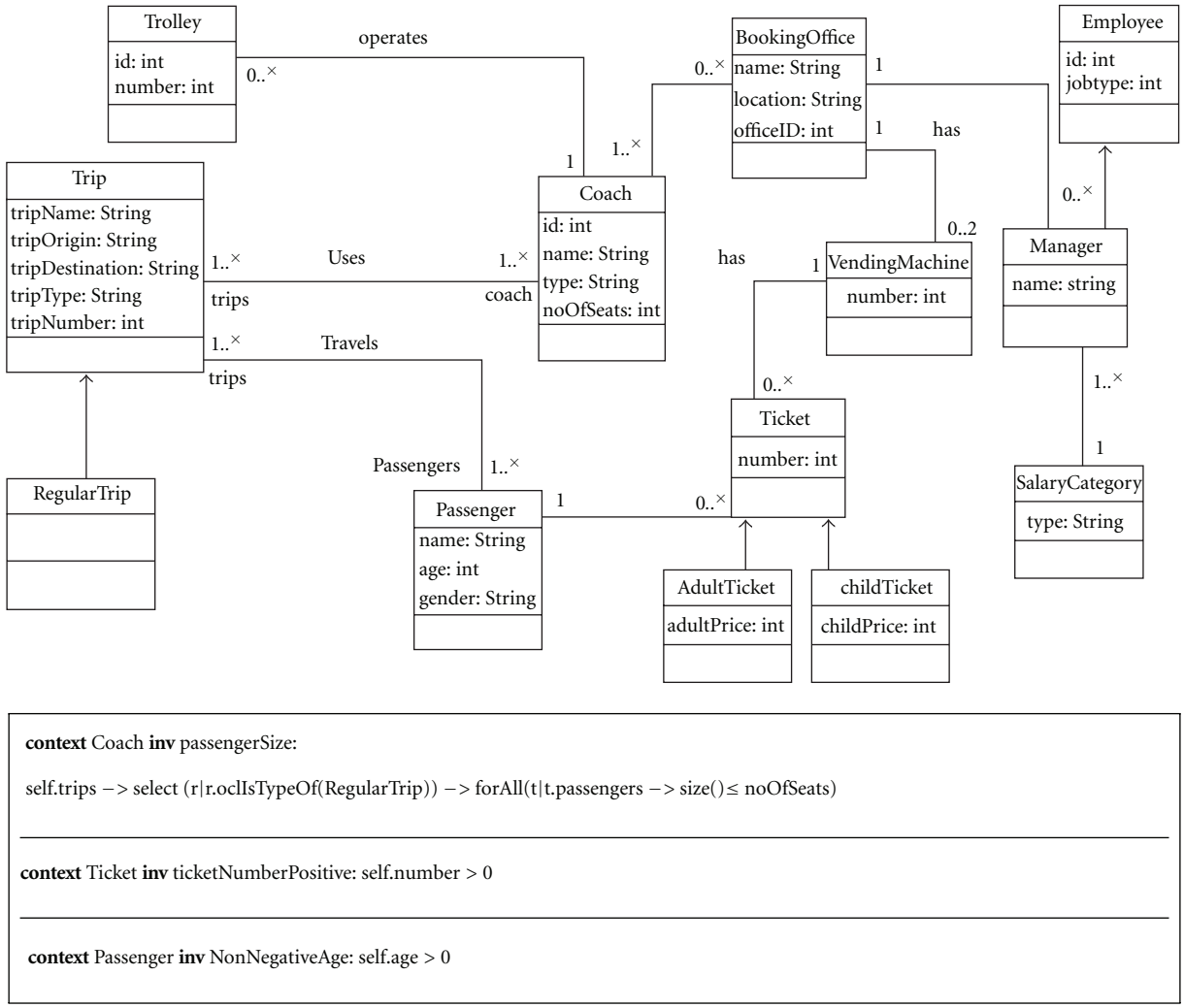


Figure 1: UML/OCL class diagram used as running example (model Coach).

all classes are not instantiated due to multiple inheritance, composition, and aggregation. In this case, the model will be considered as unsatisfiable in the case of strong satisfiability. Consequently, strong satisfiability requires the existence of an object for each concrete subclass of an abstract class.

The proposed approach instantiates objects for verification purposes based on a given class diagram and OCL constraints of the system. A successful verification result ensures that the model complies with the system specifications imposed at the start of the development phase, and, therefore, the developers may continue with transforming the model into software code.

The algorithm takes a UML/OCL model as an input, breaks it into several submodels with respect to invariants, and verifies the properties of each constraint to determine whether the input class diagram has legal instances which satisfy all integrity constraints of class attributes. The slicing algorithm can be applied over a large model to reduce the size and complexity of the UML/OCL model, so that it can be verified more efficiently. Slicing of UML class diagrams is dependent on the OCL constraints. Thus, if there are 3 constraints in the model, slicing might result in three submodels.

A *slice S* of a UML class diagram *D* is another valid UML class diagram where any element (class, association, inheritance, aggregation, etc.) appearing in *S* also appears in *D*, but the reverse does not necessarily hold.

In the context of satisfiability, saying that "a class *X* depends on a class *Y*" means that creating an object of class *Y* creates an obligation that must be satisfied by class *X*, for example, the existence of *n* corresponding objects in class *X*. Relationships like associations, aggregations, and inheritance hierarchies can create these types of dependencies. For instance, in associations the dependency is typically bidirectional, as the multiplicity of each association end imposes a dependency on the other class.

3.1. The UOST Process. The method introduced for computing UML/OCL slicing is shown in Figure 2. The process begins in step 1 by identifying the classes, associations,

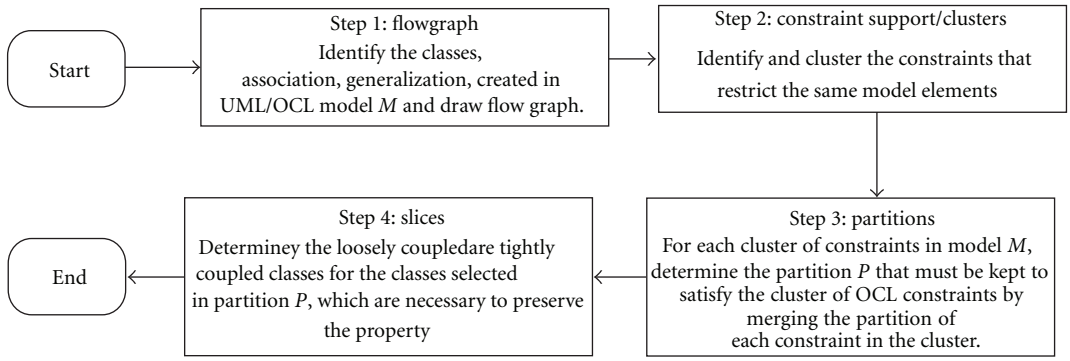


Figure 2: UOST process steps.

and generalizations created in model M and subsequently drawing a flowgraph. In step 2, we identify OCL invariants and group them if they restrict the same model elements. We call this "clustering of constraints" (constraint support). The constraint support defines the scope of a constraint. The support information can be used to partition a set of OCL invariants into a set of independent *clusters* of constraints, where each cluster can be verified separately. The following procedure is used to compute the clusters:

(i) compute the constraint support of each invariant;

(ii) keep each constraint in a different cluster;

(iii) select two constraints x and y with nondisjoint constraint supports and located in different clusters and merge those clusters;

(iv) repeat the previous step until all pairs of constraints with nondisjoint constraint supports belong to the same cluster.

In step 3, for each cluster of constraints in model M, the partition P is determined that holds all those classes and relationships restricted by the constraints in the cluster. In this step, we can capture the possible number of slices with the consideration of OCL invariants. Each partition will be a subset of the original model.

In step 4, tightly coupled classes are added to each partition in accordance with the lower bound ≥ 1 association. It means that if the constraint is restricted from class X, it is necessary to check the lower bound ≥ 1 associated classes with X. In this step, all the associated classes are added to a partition, which results in a model slice.

3.2. Flowgraph Creation: Step 1. In this section and the next, we illustrate the UOST slicing Algorithm 2 through an example. Consider the "model Coach" scenario whose UML class diagram and OCL constraints are shown in Figure 1. There are three constraints that restrict the classes, and out of them two are local invariants and one is global. An invariant is called local to a class C if it can be evaluated by examining only the values of the attributes in one object of class C. However, expressions that do not fit into this category, because they need to examine multiple objects of the same class or some objects from another class, are called global.

By applying step 1 (Figure 2), we build a flowgraph based on the identification of classes, associations, and generalizations as shown in Figure 3. We use the concept of a flowgraph to capture the dependencies among model elements. This concept is also used by other slicing approaches [29–31]. A flowgraph is a set of vertices and directed arcs where the vertices represent classes from a class diagram and the arcs model relationships between these classes. In our approach, a flowgraph contains vertices and arcs for each pair of classes connected by associations, generalizations, aggregations, or compositions. We consider two types of relationships among classes: tightly associated and loosely associated classes. These relationships attempt to capture the necessity of creating instances of one class when an instance of the other exists. Loosely coupled classes have an association with a lower bound of 0 (e.g., $0 \cdots 3$); this means if an object of class A is instantiated, then it is not necessary that an object of class B must be instantiated. Tightly coupled classes are the inverse of loosely coupled classes; that is, they have an association with a lower bound greater than 1 (e.g., $1 \cdots *$).

In the case of aggregation, composition, and generalized classes, we count them as tightly coupled classes. To differentiate aggregation, composition, and generalized classes from associations in the flowgraph, we use a solid undirected edge (—) as a shortcut for two directed arcs between the two classes. A tightly coupled association between two classes is shown as a solid arc (⟶), while a loosely coupled association is shown as a dashed arc (⇢). Table 7 briefly summarizes the criteria to assign loosely coupled and tightly coupled relationships, and Algorithm 1 shows the steps that compute a flowgraph for a given class diagram.

3.3. Applying UOST: Step 2, 3, and 4. In this section, we compute constraint support, partitions, and build the final slices for verifiability.

Considering the model Coach where *Model M (Coach, Trolley, Booking Office, Passenger, Ticket, Trip, RegularTrip, VendingMachine, Manager, Employee, SalaryCategory, AdultTicket, and ChildTicket)* and *Constraints C (passengerSize, ticketNumberPositive, and NonNegativeAge)*. We are supposed to find the legal instances of three invariants, that is, passengerSize, ticketNumberPositive, and NonNegativeAge.

Input: A model M
Output: A labeled directed graph $G = \langle V, E \rangle$

(1) {Start with the empty graph}
(2) Let $V \leftarrow \emptyset$ and $E \leftarrow \emptyset$
(3) {Add all classes of the model to the flowgraph}
(4) **for** class c in model M **do**
(5) $V \leftarrow V \cup \{c\}$
(6) **end for**
(7) {Create incoming and outgoing arcs in the flowgraph}
(8) **for** each association end A in model M **do**
(9) $E \leftarrow (x, y)$ where x is the type of the association end and y is the type of the other class in the association
(10) **if** the lower bound of the multiplicity of A is ≥ 1 **then**
(11) Label the arc (x, y) as tightly coupled
(12) **else if** the lower bound of the multiplicity of $A = 0$ **then**
(13) Label the arc (x, y) as loosely coupled
(14) **end if**
(15) **end for**
(16) **for** each generalization, aggregation and composition G between classes x and y **do**
(17) $E \leftarrow E \cup \{(x, y)\} \cup \{(y, x)\}$
(18) Label the arcs (x, y) and (y, x) as tightly coupled
(19) **end for**

Algorithm 1: Flowgraph creation.

Input: Property being verified
Output: A partition P of the model M into non-necessarily disjoint submodels

(1) $G \leftarrow BuildFlowGraph(M)$ {Creating the flowgraph}
(2) {Cluster the OCL constraints}
(3) **for** each pair of constraints $c1, c2$ in M **do**
(4) **if** ConstraintSupport$(M, c1) \cap$ ConstraintSupport$(M, c2) \neq \emptyset$ **then**
(5) MergeInSameCluster$(c1, c2)$
(6) **end if**
(7) **end for**
(8) {Work on each cluster of constraints separately}
(9) **for** each cluster of constraints Cl **do**
(10) subModel $\leftarrow$ empty model {Initialize the subModel to be empty}
(11) {Initialize worklist}
(12) workList $\leftarrow$ Union of the ConstraintSupport of all constraints in the cluster
(13) **while** workList not empty **do**
(14) $node \leftarrow$ first(workList) {Take first element from workList and remove it}
(15) workList $\leftarrow$ workList $\setminus$ $node$
(16) **for** each subclass or superclass c of $node$ **do**
(17) subModel $\leftarrow$ subModel $\cup \{c\}$
(18) **if** c was not before in the subModel **then**
(19) workList $\leftarrow$ workList $\cup \{c\}$
(20) **end if**
(21) **end for**
(22) **for** each class c tightly coupled to $node$ **do**
(23) **if** $Property$ = weak SAT **then**
(24) subModel $\leftarrow$ subModel $\cup \{c\}$
(25) **else if** $Property$ = strong SAT **then**
(26) workList $\leftarrow$ workList $\cup \{c\}$
(27) **end if**
(28) **end for**
(29) **end while**
(30) **end for**

Algorithm 2: Slicing algorithm.

Table 7: Loosely and tightly coupled classes.

UML relationship	*Loosely/tightly coupled*	*Arc/Edge*
Association: lower bound ≥1 (e.g., $1 \cdots *$)	Tightly coupled	⟶
Association: lower bound = 0 (e.g., $0 \cdots 3$)	Loosely coupled	⇢
Generalization, aggregation, and composition	Tightly coupled	—

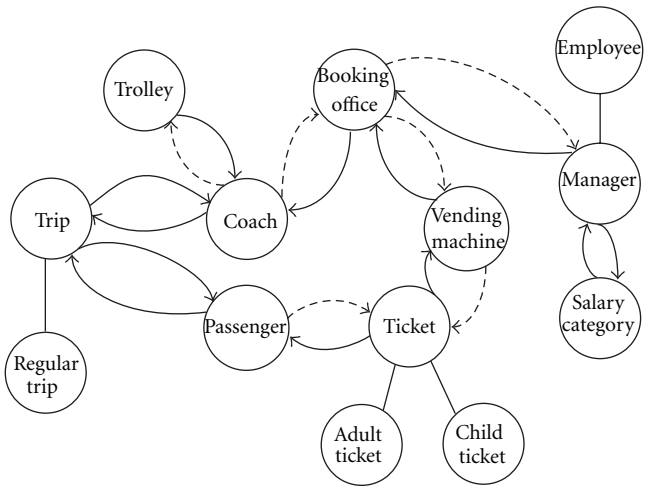


Figure 3: Flowgraph of the model Coach.

Applying step 2, we identify and cluster the OCL constraints. It is necessary to cluster the invariants beforehand, as the set of model elements constrained by each invariant may have an interaction. Considering Figure 1, there are three invariants that restrict class Coach, Ticket, and Passenger. In this case, constraint *NonNegativeAge* will be merged with *passengerSize* because the properties of these constraints can be satisfied from similar model elements. Meanwhile, the properties of *ticketNumberPositive* can be satisfied from different model elements.

In step 3, for each constraint and group of constraints in model M, the partition P that holds all those classes and multiplicities from which the cluster of invariants are constrained will be determined. In this step, we can capture the possible number of slices with the consideration of OCL invariants. Each partition will be a subset of the original model.

In step 4, all the tightly coupled classes are added into formed partitions in order to preserve the property of an invariant because it is necessary that the object of each class must be instantiated in case there is strong satisfiability; otherwise, the property will not be satisfied. For the cluster of passengerSize and NonNegativeAge, we need classes *Coach, Trip, RegularTrip, and Passenger* while classes *Ticket, BookingOffice, Trolley, VendingMachine, Manager, Employee, SalaryCategory, AdultTicket, and ChildTicket* can safely be removed from the slice (i.e., s1).

Similarly, to satisfy the properties of ticketNumberPositive, we require classes *BookingOffice, Coach, Trip, RegularTrip, Passenger, VendingMachine, Ticket, AdultTicket, and ChildTicket*, while classes *Trolley, Manager, Employee, and SalaryCategory* can be deleted from the slice (i.e., s2). Figures 4(a) and 4(b) highlight the final slices passed to the verification tool for strong satisfiability. The members of a slice are hence defined as follows:

(i) the classes and relationships in the cluster of constraint supports are part of the slice;

(ii) any class with a tightly coupled relationship to a class in the slice is also a part of the slice, as is the relationship.

4. Nondisjoint Solution

In this section, we present the solution that still preserves the satisfiability in case of nondisjoint submodels. Nondisjoint submodels may occur if a common class is used in several constraints. In the worst case, the clustering technique in Section 3 may result in the whole UML model and consequently no improvements in verification time. The nondisjoint solution can be selected by the designer in the tool (UMLtoCSP) if the model is constrained by several invariants in a way which makes clustering ineffective. The nondisjoint solution differs from the UOST process (see Figure 2) in that it works without clustering the model elements, hence making it still possible to improve verification time.

The nondisjoint solution is defined as follow.

Let C be a set of classes, and let $A = \bigcup_{c \in C} A_c$ be the set of attributes. $M = C$ is the model consisting of these classes. Let R be the set of binary associations among two classes. Each association R is defined as a tuple $(C_1, C_2, m_1, M_1, m_2, M_2)$, where

(i) $C_1 \in C$ is a class

(ii) $C_2 \in C$ is a class

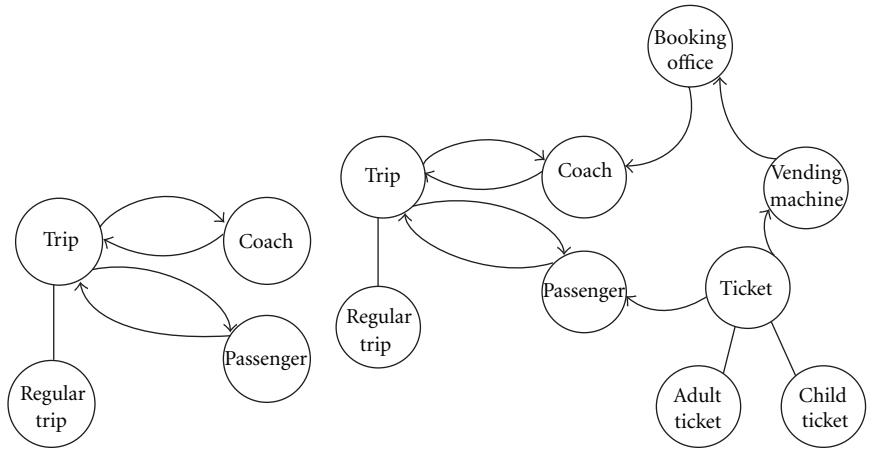
(iii) m_1 and m_2 are nonnegative integers $\in Z^+$, where m_1 and m_2 correspond to the lower bound of the multiplicity of each association end for C_1 and C_2, respectively

(iv) M_1 and M_2 are nonnegative integers or infinity ($M_1 \in (Z^+ \bigcup \{\infty\})$), where M_1 and M_2 correspond to the upper bound of the multiplicity of each association end for C_1 and C_2, respectively, and $M_i \geq m_i$.

A model M can be defined as a tuple: (C, A, R). A submodel S of model $M = (C, A, R)$ is another model (C', A', R') such that

(i) $C' \in C$,

(ii) $R' \in R$,



(a) Slice for passengerSize and NonNegativeAge (s1)

(b) Slice for ticketNumberPositive (s2)

Figure 4: Slices s1 and s2 of model Coach.

(iii) $A' \in A$,

(iv) $c \in C' \rightarrow A_c \subseteq A'$,

(v) $(C_1, C_2, m_1, M_1, m_2, M_2) \in R' \rightarrow C_1, C_2 \in C'$.

An OCL expression specifies the model entity for which the OCL expression is defined. CL represents the OCL invariants while CL_c are the clusters of constraints. The work list is defined as W_L which is the union of the constraint support of all constraints in the cluster.

(i) *Satisfiability (Strong/Weak)* If the objects of a given class C in a submodel S are instantiated as per given expression in the cluster of OCL constraints CL_c, then submodel S is satisfiable.

(ii) *Unsatisfiability* If there are two or more constraints whose interaction is unsatisfiable, then submodel S is also unsatisfiable. It indicates that some expression in the OCL invariant is violated and that the objects of the classes cannot be instantiated according to the given OCL expression.

A class diagram can be unsatisfiable due to several reasons. First, it is possible that the model provides inconsistent conditions on the number of objects of a given type. Inheritance hierarchies, multiplicities of association/aggregation ends, and textual integrity constraints (e.g., Type::allInstances() → size() = 7) can restrict the possible number of objects of a class. Second, it is possible that there are no valid values for one or more attributes of an object in the diagram. Within a model, textual constraints provide the only source of restrictions on the values of an attribute, for example, self.x = 7. Finally, it is possible that the unsatisfiability arises from a combination of both factors; for example, the values of some attributes require a certain number of objects to be created, which contradicts other restrictions.

To sum up, an unsatisfiable model either contains an unsatisfiable textual or graphical constraint or an unsatisfiable interaction between one or more textual or graphical constraints; that is, the constraints can be satisfied on their own but not simultaneously.

In a class diagram, there could be a possibility to have one or more relationships between two classes; that is, a class may have a relationship with itself and there may be multiple relationships between two classes. Multiple links between two classes or a link from one class to itself is called a "cycle." For example, a cycle exists between "Researcher" and "Paper" in Figure 5. The "maximum label" is the highest upper-bound multiplicity of the associations in a cycle. For example, the maximum label is 1 for constraints restricting papers and 3 for constraints restricting researchers.

Any cycle in the class diagram where the maximum label is 1 is inherently satisfiable, and it will be called *safe*. However, cycles where the maximum label ≥2 can be unsatisfiable. Such cycles will be called *unsafe*. By "safe" we mean any cycle where the maximum label is 1 and imposing a single constraint is inherently satisfiable where the OCL expression is `self.attrib op expression` where *attrib* is an attribute of a basic type (Boolean, Integer, Float, String) not constrained by any other constraint, *op* is a relational operator $(=, \neq, <, >, \leq, \geq,)$ and *expression* is a "safe" OCL expression which does not include any reference to *attrib*. The safe expression is a side-effect-free expression which cannot evaluate to the undefined value in OCL (`OclUndefined`). This means that we do not allow divisions that can cause a division-by-zero or collection operations which are undefined on empty collections like first().

We present the nondisjoint solution if slicing is applied over a UML model without clustering the constraints (i.e., without step 2 in the UOST process).

There are three major steps that need to be considered as a solution:

(i) find the common class in all slices of the model (M);

(ii) for each constraint, find the maximum of the lower-bound (m_1) multiplicities relevant to the constraint from all associations of the common class. Set this

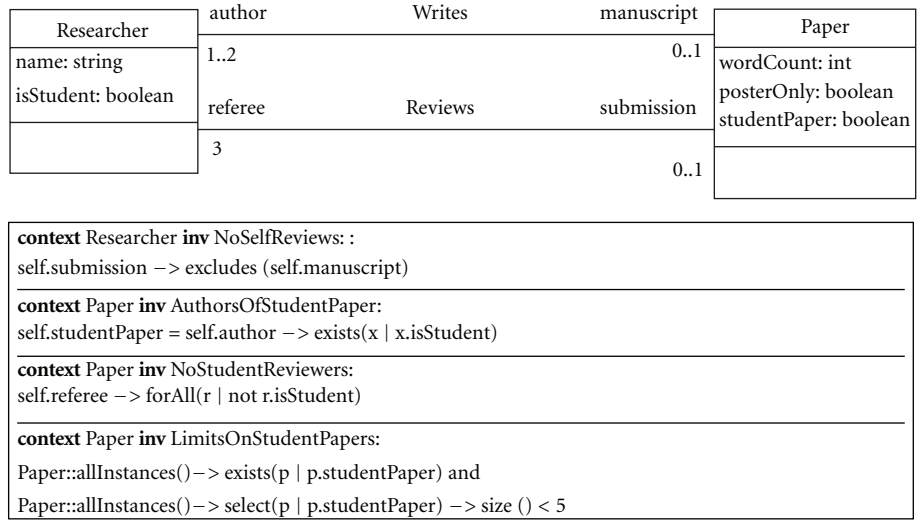


Figure 5: UML/OCL class diagram of "Paper-Researcher" [32].

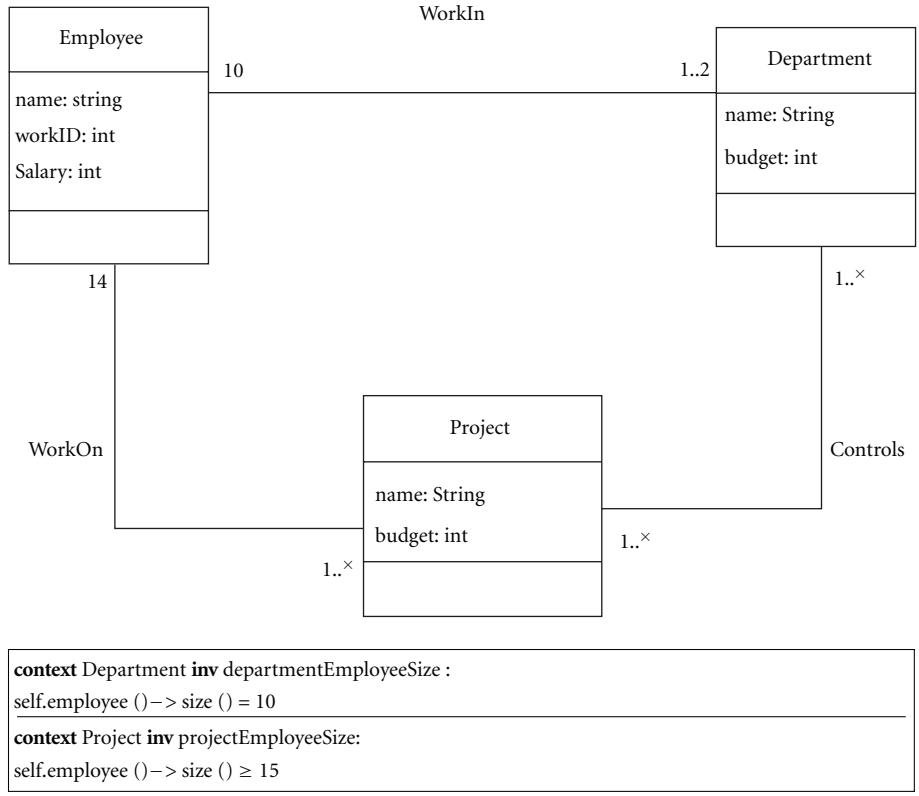


Figure 6: UML/OCL class diagram used as nondisjoint solution (Company model).

maximum as the base value. $\text{Base}_c = \max(m_1)$, where $(c, C_2, m_1, M_1, m_2, M_2) \in R$;

(iii) compare the base value using the expression given in each constraint.

The OCL constraints can be either textual OCL invariants or graphical restrictions like multiplicities of association ends. This property is important not only because it can point out inconsistent models, but also because it can be used to check other interesting properties like the redundancy of an integrity constraint. For example, there could be a case where the invariants are constrained from the same class of the model. Figure 6 introduces a class diagram of "Company model" used to exemplify our nondisjoint solution. There are two constraints, *departmentEmployeeSize,* and *projectEmployeeSize* whose properties need to be checked. Invariant

departmentEmployeeSize is satisfiable; however, invariant *projectEmployeeSize* is unsatisfiable due to a violation of multiplicity. After applying the slicing technique without clustering the invariants, we will receive two submodels, that is, two nondisjoint slices. Slice 1 will consist of class "Department" and class "Employee" for constraint *departmentEmployeeSize*. Similarly, class "Project" and class "Employee" for invariant *projectEmployeeSize* will be part of slice 2.

In this case, slice 1 is satisfiable; however, slice 2 is unsatisfiable. The definition of the slicing procedure ensures that the property under verification is unsatisfiable after partitioning because the overall interaction of the model is unsatisfiable.

Initially, our nondisjoint approach finds the common class in all slices of model (M), that is, class "Employee." Secondly, the method finds the maximum of minimum (max_min) multiplicities from the common class (Employee) for each constraint considering its navigation. For example, the navigation of invariant "departmentEmployeeSize" is class "Department" navigating to class "Employee". Therefore, the approach considers the multiplicity between the navigation of class department and class employee, that is, "10" and "$1 \cdots 2$". As the constraint restricts class employee, "10" is the base value for the "departmentEmployeeSize" invariant. Similarly, "14" is the base value for the navigation of class "Project" and class "Employee."

Finally, the method compares the base value (i.e., 10) for invariant "departmentEmployeeSize" using the expression given in a constraint self.employee() $\rightarrow$ size() = 10 whose interaction is satisfiable. However, invariant "projectEmployeeSize" is violating the condition, that is, using the expression self.employee() $\rightarrow$ size() ≥ 15, where 14 is not ≥ 15. Hence, the overall interaction of the model is unsatisfiable.

5. Empirical Study

This section presents the speedup achieved by slicing in two of the tools, that is, UMLtoCSP [4] and Alloy [6]. We have developed prototype implementations of the slicing procedure in UMLtoCSP and Alloy to conduct these experiments. The goal of the empirical study is to show the achieved speedup in the verification process before and after slicing. Therefore, the general question addressed here is

"*How can we improve the efficiency of the verification process for complex UML/OCL class diagrams?*"

5.1. UOST Implementation in UMLtoCSP. We have implemented our proposed slicing technique (UOST) in UMLtoCSP [4] in order to show the improvement of the efficiency in the verification process. After developing UOST, we renamed the tool to UMLtoCSP (UOST). The execution time of verification of an original UMLtoCSP depends mainly on the number of classes/attributes and the parameters offered during the transformation to the constraint satisfaction problem (CSP). In case of small models, UMLtoCSP provides quick results while for larger ones, the tool takes a huge of amount of time. In order to evaluate the efficiency

Table 8: Description of the examples.

Example	Classes	Associations	Attributes	Invariants
Paper-Researcher	2	2	6	1
Coach	15	12	2	2
DBLP	17	27	38	26
Tracking System	50	60	72	5
Script 1	100	110	122	2
Script 2	500	510	522	5
Script 3	1000	1010	1022	5

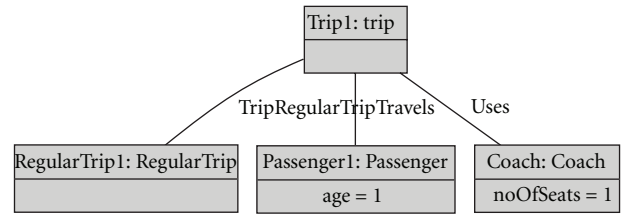


Figure 7: Submodel 1 (s1) object diagram for strong satisfiability.

of our developed UOST approach in UMLtoCSP, several models have been used (Table 8). UMLtoCSP takes a lot of time to verify the instances of large examples; therefore, we set a time out to 2 hours which is equal to 7200 seconds. If UMLtoCSP does not verify the model in the prescribed time, we will count this situation as *Time out*.

Table 9 summarizes the experimental results obtained by UMLtoCSP and UMLtoCSP (UOST) running on an Intel Core 2 Duo 2.10 Ghz with 2 Gb of RAM, where the column OVT is the original verification time of UMLtoCSP, column TVT is the total verification time of all slices of UMLtoCSP (UOST), and column speedup shows the efficiency obtained after the implementation of the slicing approach. We have used the following parameters for the experiments: each class may have at most 4 instances, associations may have at most 1010 links, and attributes may range from 0 to 1022. The speedup is calculated using the equation below:

$$\left\{1 - \left(\frac{\text{TVT}}{\text{OVT}}\right)\right\} * 100. \quad (1)$$

Figure 7 shows the object diagram for s1 in the case of strong satisfiability, and Figure 8 represents the object diagram for s2 in the case of weak satisfiability, where there is no need to instantiate unused subclasses (i.e., AdultTicket and ChildTicket). The object diagrams are generated using UMLtoCSP (UOST).

As a conclusion, the process of slicing is fast even for large and complex models having hundreds of classes. However, this effectiveness depends primarily on the specific type of models being considered. As such, small models and models where UMLtoCSP already performed well gain little from slicing. Similarly, models with no unconstrained attributes and all classes and constraints being interdependent gain little advantage under this technique. In the worst case, the verification time with slicing is the same as that without

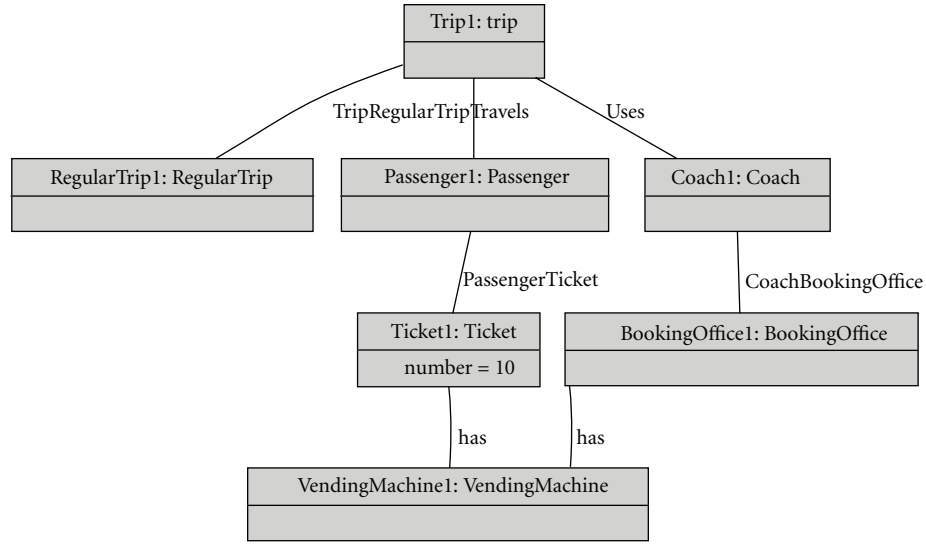


Figure 8: Submodel 2 (s2) for weak satisfiability.

Table 9: Description of experimental results (UMLtoCSP).

Before slicing (UMLtoCSP)			After slicing (UMLtoCSP UOST)				
Classes	Attributes	OVT	Attributes	ST	SVT	TVT	Speedup %
2	6	0.04 s	3	0.00 s	0.040 s	0.040 s	0%
15	2	5008.76 s	0	0.00 s	0.178 s	0.178 s	99.99%
17	38	Time out	6	0.020 s	0.432 s	0.452 s	N/A
50	72	3605.35 s	55	0.016 s	0.031 s	0.047 s	99.99%
100	122	Time out	117	0.016 s	0.032 s	0.048 s	N/A
500	522	Time out	502	0.062 s	0.028 s	0.090 s	N/A
1000	1022	Time out	1012	0.282 s	0.339 s	0.621 s	N/A

OVT: Original Verification Time. ST: Slicing Time. SVT: Sliced Verification Time. TVT: Total Verification Time.

slicing. In contrast, dramatic improvements in speed-up by the order of several magnitudes are observed where slicing manages to partition the model and abstract attributes. The tiny overhead introduced by slicing and the tool-independent nature of this approach are additional reasons in favor of adding slicing to existing formal verification toolkits.

5.1.1. DBLP Conceptual Schema in UMLtoCSP. In this section, we have applied our slicing technique to the digital bibliography and library project (DBLP) structural schema modeled as a UML class diagram [33]. It is a computer science bibliographical website and has existed since the 1980s. The DBLP structural schema deals with people and their publications, which can be edited books and authored publications. The class diagram has 17 classes and 27 integrity constraints. This case study is interesting for our problem since it has complex invariants and is a real-world case study. Therefore, we intend to apply our slicing approach to this DBLP case study in order to show that our methods work upon external case studies and can improve the efficiency of the verification process. After applying the technique, two submodels are received: submodel 1 consists of 10 classes annotated with 8 OCL constraints and submodel 2 comprises of 2 classes annotated with 2 OCL constraints. UMLtoCSP did not verify the DBLP UML/OCL class diagram even in two hours of time due to the scope of invariants. However, with the slicing technique, the same model is now verifiable in just 0.452 seconds. Table 9 summarizes the experimental results for DBLP conceptual schema. We have achieved a 99% speedup for the verification of the "DBLP" class diagram.

5.2. Limitations. Our proposed technique is limited in scope and cannot partition the UML/OCL class diagram and abstract the attributes if the constraints are restricted to all classes using all attributes of the class diagram. In this case, the technique will consider a UML/OCL class diagram as a single model, and, therefore, there will be no difference between UMLtoCSP and UMLtoCSP (UOST). Table 10 describes the worst case examples that cannot be sliced using UOST. The example *Paper-Researcher* is a real-world example which contains 2 classes, 6 attributes, 2 associations, and 4 invariants while example *Company* is script generated and has 100 classes, 100 attributes, 100 associations, and 100 invariants. In these examples, partitioning cannot be done

Table 10: Worst-case examples.

Tool and example	Classes	Associations	Attr	Inv	NOS	OVT
UMLtoCSP (Paper-Researcher)	2	2	6	4	0	0.040 s
UMLtoCSP (UOST) (Paper-Researcher)	2	2	6	4	0	0.040 s
UMLtoCSP (Company)	5	7	3	5	0	0.070 s
UMLtoCSP (UOST) (Company)	5	7	3	5	0	0.070 s
UMLtoCSP (Script)	100	110	122	100	0	Time out
UMLtoCSP (UOST) (Script)	100	110	122	100	0	Time out

Attr: Associations. Inv: Invariants. NOS: Number of Slices. OVT: Original Verification Time.

Table 11: Description of the examples.

Example	Classes	Associations	Attributes	Invariants
Paper-Researcher	2	2	6	1
Coach	15	12	2	2
DBLP	17	27	38	26
Tracking System	50	60	72	5
Script 1	100	110	122	2
Script 2	500	510	522	5
Script 3	1000	1010	1022	5

by the proposed UOST technique because each instance of a class is constrained by an invariant.

5.3. UOST Implementation in Alloy. In this section, we present several examples in the Alloy specification in order to prove that our developed slicing technique is neither tool dependent nor formalism dependent. We compare the verification time of several UML/OCL class diagrams using the Alloy analyzer with and without the UOST technique. Table 11 describes the set of benchmarks used for our comparison: the number of classes, associations, invariants, and attributes. The benchmarks "Script" was programmatically generated, in order to test large-input models. Of these models, we consider the "Script" models to be the best possible scenarios for slicing (large models with many attributes and very few constraints).

Tables 12, 13, 14, 15, 16, and 17 summarize the experimental results obtained using the Alloy analyzer before and after slicing, running on an Intel Core 2 Duo Processor 2.1 Ghz with 2 Gb of RAM. Each table represents the results as described in the benchmark (Table 11). The execution time is largely dependent on the defined scope; therefore, in order to analyze the efficiency of verification, the scope is limited to 7. The Alloy analyzer will examine all the examples with up to 7 objects and try to find one that violates the property. For example, scope 7 means that the Alloy analyzer will check models whose top level signatures have up to 7 instances.

All times are measured in seconds (s). For each scope (before slicing), the translation time (TT), solving time (ST), and the summation of the TT and ST, which is the total execution time, are described. Similarly, for each scope (after slicing) we measure the sliced translation time (STT), sliced solving time (SST), and the summation of STT and SST. Similarly, the column speedup shows the efficiency obtained after the implementation of the slicing approach.

Previously with no slicing, it took 0.820 s (scope 7) for the execution of the "Tracking System" and 282.161 s (scope 7) for "Script 3." Using the UOST approach, it takes only 0.058 s (scope 7) for "Tracking System" and 0.052 s (scope 7) for "Script 3." It is an improvement of 93% and 99.98%, respectively. In addition, the improvement can also be achieved for larger scopes as well. For instance, results for up to scope 50 can be achieved for the "Tracking System" and scope 35 for "Script."

5.3.1. DBLP Conceptual Schema in Alloy. We have programmed DBLP in Alloy in order to show the results in an external tool with a more real-world example. The specification of the class diagram is the same as above. The execution time in Alloy is largely dependent on the defined scope. We already explored the results with smaller scopes; however, in this section we present experimental results (with slicing and without slicing) with larger scopes, that is, 15–19. After applying the technique, two submodels are received: submodel 1 consists of 10 classes annotated with 8 OCL constraints and submodel 2 comprises of 2 classes annotated with 2 OCL constraints. Table 18 summarizes the experimental results for DBLP conceptual schema where we have achieved maximum 57% improvement for a real-world case study. The percentage is calculated using the following equation:

$$\left\{1 - \left(\frac{\text{TT} + \text{ST}}{\text{STT} + \text{SST}}\right)\right\} * 100. \quad (2)$$

5.4. Statistical Comparison. In this section, we present some statistical comparisons of the various experiments comparing the verification time before and after slicing using UMLtoCSP (Table 9) and Alloy (Tables 12–18). We have included the mean verification time and the standard deviation for both "before slicing" and "after slicing" cases. Table 19 briefly summarizes the results.

6. Related Work

In this section, we discuss existing work on model partitioning or slicing. Most of the work in this area is done

Table 12: Slicing results in Alloy for Paper-Researcher example.

	Before slicing			After slicing			
Scope	TT	ST	TT + ST	STT	SST	STT + SST	Speedup %
2	0.003 s	0.009 s	0.012 s	0.003 s	0.005 s	0.008 s	34%
3	0.007 s	0.008 s	0.015 s	0.003 s	0.006 s	0.009 s	40%
4	0.012 s	0.008 s	0.020 s	0.004 s	0.006 s	0.010 s	50%
5	0.017 s	0.010 s	0.027 s	0.004 s	0.009 s	0.013 s	52%
6	0.016 s	0.015 ms	0.031 s	0.005 s	0.009 s	0.014 s	55%
7	0.019 s	0.015 ms	0.034 s	0.006 s	0.009 s	0.015 s	56%

TT: Translation Time. ST: Solving Time. STT: Sliced Translation Time. SST: Sliced Solving Time.

Table 13: Slicing results in Alloy for Coach example.

	Before slicing			After slicing			
Scope	TT	ST	TT + ST	STT	SST	STT + SST	Speedup %
2	0.007 s	0.010 s	0.017 s	0.003 s	0.005 s	0.008 s	53%
3	0.014 s	0.019 s	0.033 s	0.005 s	0.008 s	0.013 s	61%
4	0.028 s	0.020 s	0.048 s	0.007 s	0.010 s	0.017 s	62%
5	0.036 s	0.031 s	0.067 s	0.012 s	0.015 s	0.027 s	65%
6	0.045 s	0.050 s	0.095 s	0.017 s	0.015 s	0.032 s	67%
7	0.081 s	0.077 s	0.158 s	0.034 s	0.017 s	0.051 s	68%

TT: Translation Time. ST: Solving Time. STT: Sliced Translation Time. SST: Sliced Solving Time.

for UML architectural models, model slicing, and program slicing which is limited to slicing only. Their goal of slicing is to break larger programs or models into small submodels to reuse the required segments. However, research work on partitioning of UML/OCL class diagrams in terms of verifiability is not found in the literature. Previously, we proposed a slicing technique for models considering a UML class diagrams annotated with unrestricted OCL constraints and a specific property for verification [15, 16]. The slicing approach was based on disjoint slicing, clustering, and the removal of trivially satisfiable constraints. An implementation of the slicing technique has been developed in a UMLtoCSP tool. Experimental results demonstrate that slicing can verify complex UML/OCL models and speed up the verification time.

In contrast, this paper presents an improved slicing technique which can still preserve the property under verification for nondisjoint set of submodels. We have also demonstrated results in an external tool, "Alloy," in order to prove that the proposed slicing technique is not limited to a single tool (i.e., UMLtoCSP) but can also be used for other formal verification tools. The slicing procedure breaks the original model into submodels (slices) which can be verified independently and where irrelevant information has been abstracted. The definition of the slicing procedure ensures that the property under verification is preserved after partitioning.

6.1. UML Model Slicing. The theory of *model slicing* to support and maintain large UML models is mostly discussed in the literature. Current approaches of model verification have an exponential worst-case runtime. Context-free slicing of the model summarizes static and structural characteristics of a UML model. The term context points towards the location of a particular object. It takes into account static and structural aspects of a UML model and excludes the enclosure of interaction information [34]. Similarly, to compute a slice of a class hierarchy of a program, it is necessary to eliminate those slices that are unnecessary thereby ensuring that the behavior of the programs would not be affected. This approach represents the criteria of model abstraction [35].

One possible approach to manage the complexity of the UML metamodel is to divide the metamodel into a set of small metamodels for each discussed UML diagram type [36]. The proposed method defines a metamodel of a directed multigraph for a UML Metamodel Slicer. The slicer builds sub-metamodels for a diagram with model elements. Another slicing technique for static and dynamic UML models presents the transformation of a UML architectural model into a model dependency graph (MDG). It also merges a different sequence of diagrams with relevant information available in a class diagram [37].

6.2. Architectural Slicing. The concept of *architectural slicing* is used to remove irrelevant components and connectors, so that the behavior of the slice is preserved [38]. This research introduces a new way of slicing. Architectural slicing is used to slice a specific part of a system's architecture. The sliced part is used to view high-level specifications. Similar to this approach, a dependency analysis technique is developed which is based on the slicing criteria of an architectural specification as a set of component parts [28]. The technique is named chaining. It supports the development of software architecture by eliminating unnecessary parts of the system.

Table 14: Slicing results in Alloy for Tracking System.

	Before slicing			After slicing			
Scope	TT	ST	TT + ST	STT	SST	STT + SST	Speedup %
2	0.020 s	0.046 s	0.066 s	0.005 s	0.008 s	0.013 s	81%
3	0.083 s	0.091 s	0.174 s	0.009 s	0.011 s	0.020 s	89%
4	0.096 s	0.185 s	0.254 s	0.013 s	0.011 s	0.024 s	90%
5	0.158 s	0.173 s	0.332 s	0.020 s	0.012 s	0.032 s	90%
6	0.233 s	0.367 s	0.600 s	0.025 s	0.023 s	0.048 s	92%
7	0.325 s	0.495 s	0.820 s	0.030 s	0.028 s	0.058 s	93%

TT: Translation Time. ST: Solving Time. STT: Sliced Translation Time. SST: Sliced Solving Time.

Table 15: Slicing results in Alloy for Script 1.

	Before slicing			After slicing			
Scope	TT	ST	TT + ST	STT	SST	STT + SST	Speedup %
2	0.110 s	0.133 s	0.243 s	0.007 s	0.009 s	0.016 s	93%
3	0.161 s	0.290 s	0.451 s	0.009 s	0.009 s	0.018 s	96%
4	0.224 s	0.591 s	0.815 s	0.014 s	0.012 s	0.026 s	97%
5	0.349 s	0.606 s	0.955 s	0.017 s	0.016 s	0.033 s	97%
6	0.589 s	1.077 s	1.666 s	0.027 s	0.025 s	0.052 s	97%
7	0.799 s	1.392 s	2.191 s	0.038 s	0.025 s	0.063 s	97%

TT: Translation Time. ST: Solving Time. STT: Sliced Translation Time. SST: Sliced Solving Time.

Table 16: Slicing results in Alloy for Script 2.

	Before slicing			After slicing			
Scope	TT	ST	TT + ST	STT	SST	STT + SST	Speedup %
2	1.839 s	3.021 s	4.860 s	0.006 s	0.007 s	0.013 s	99.7%
3	2.567 s	7.489 s	10.056 s	0.011 s	0.008 s	0.019 s	99.8%
4	3.374 s	8.320 s	11.694 s	0.014 s	0.009 s	0.023 s	99.8%
5	4.326 s	21.837 s	26.163 s	0.018 s	0.014 s	0.032 s	99.8%
6	5.231 s	32.939 s	38.170 s	0.025 s	0.014 s	0.039 s	99.8%
7	6.477 s	59.704 s	66.181 s	0.035 s	0.016 s	0.051 s	99.9%

TT: Translation Time. ST: Solving Time. STT: Sliced Translation Time. SST: Sliced Solving Time.

Table 17: Slicing results in Alloy for Script 3.

	Before slicing			After slicing			
Scope	TT	ST	TT + ST	STT	SST	STT + SST	Speedup %
2	9.548 s	12.941 s	22.489 s	0.006 s	0.008 s	0.014 s	99.93%
3	9.734 s	30.041 s	39.775 s	0.013 s	0.010 s	0.023 s	99.94%
4	12.496 s	66.861 s	79.357 s	0.019 s	0.010 s	0.029 s	99.96%
5	15.702 s	85.001 s	100.703 s	0.022 s	0.013 s	0.035 s	99.96%
6	19.496 s	185.118 s	204.614 s	0.029 s	0.016 s	0.045 s	99.97%
7	23.089 s	259.072 s	282.161 s	0.035 s	0.017 s	0.052 s	99.98%

TT: Translation Time. ST: Solving Time. STT: Sliced Translation Time. SST: Sliced Solving Time.

Table 18: Slicing results in Alloy for DBLP conceptual schema.

	Before slicing			After slicing			
Scope	TT	ST	TT + ST	STT	SST	STT + SST	Speedup %
15	10.373 s	3.649 s	14.022 s	5.801 s	0.946 s	6.747 s	52%
17	15.462 s	8.838 s	24.300 s	9.568 s	1.394 s	10.962 s	55%
19	23.170 s	3.958 s	27.128 s	10.656 s	0.872 s	11.528 s	57%

TT: Translation Time. ST: Solving Time. STT: Sliced Translation Time. SST: Sliced Solving Time.

Table 19: Some statistical comparisons on empirical observations.

Type of empirical study	Mean values (in seconds)		Standard deviation	
	OVT	TVT	OVT	TVT
UMLtoCSP Verification Time (Table 9)	N/A	0.405	N/A	0.484
Alloy for Paper-Researcher (Table 12)	0.026	0.012	0.027	0.013
Alloy for Coach example (Table 13)	0.087	0.030	0.100	0.034
Alloy for Tracking System (Table 14)	0.469	0.038	0.547	0.042
Alloy for Script 1 (Table 15)	1.304	0.041	1.495	0.045
Alloy for Script 2 (Table 16)	33.695	0.034	40.977	0.037
Alloy for Script 3 (Table 17)	151.112	0.038	185.205	0.040
Alloy for DBLP Conceptual Schema (Table 18)	22.331	9.933	22.165	9.620

OVT: Original Verification Time (before slicing). TVT: Total Verification Time (after slicing).

Furthermore, the notion of dynamic software architecture slicing (DSAS) supports software architecture analysis. This work is useful when a huge amount of components is available. DSAS extracts the useful components of the software architecture [39].

6.3. Program Slicing. *Program slicing* [1, 31] techniques work on the code level, decomposing source code automatically. In this research, a dataflow algorithm is presented for program slices. A recursive program written in the Pascal language is used to compute the slices. A comparable algorithm is developed to slice the hierarchies of C++ programs. It takes C++ class and inheritance relations as an input and eliminates all those data members, member functions, classes, and relationships that are irrelevant ensuring that the program behavior is maintained. This work inspired us to reduce and eliminate those classes and relationships which do not have any relation to the UML/OCL class diagram [1].

However, to the best of our knowledge, none of the previous approaches consider OCL constrains and none is oriented towards verification of UML/OCL class diagrams. All the related work presented so far is not similar to our approach because it is based on the slicing of UML models while our proposed slicing techniques also cover verifiability of UML/OCL models. In contrast, we compute a slice that includes only those classes which are necessary to preserve in order to satisfy the OCL constraints that restrict the classes.

7. Conclusion and Future Work

This paper presents an evaluation of UML/OCL tools along with benefits and limitations. There is a common problem with the efficiency of the verification process in the UML/OCL tools. Therefore, we have further examined the problem with efficiency analysis, that is, worst-case runtime. We have proposed a slicing technique (UOST) to reduce the verification time in order to improve the efficiency of the verification process. The approach accepts a UML/OCL model as input and automatically breaks it into submodels where the overall model is satisfiable if all submodels are satisfiable. We propose to (1) discard from the model those classes that do not restrict any constraints and are not tightly coupled and (2) eliminate all attributes irrelevant for the analysis being performed. The presented approach of model slicing can ease model analysis by automatically identifying the parts of the model that are useful to satisfy the properties in the model. During the verification process, complex models require many resources (such as memory consumption and CPU time), making verification unbearable with existing tools. UOST can help reduce the verification time. We have implemented this approach in UMLtoCSP and in an external tool Alloy to provide a proof of concept.

As our future work, we plan to integrate the slicing technique in the HOL-OCL tool and in USE. So far, we have applied the slicing in verification tools, and our next step is to work on validation tools such as USE and Mova.

References

[1] G. Georg, J. Bieman, and R. B. France, "Using alloy and uml/ocl to specify run-time configuration management: a case study," in *Proceedings of UML Workshop on the Practical UML-Based Rigorous Development Methods*, Lecture Notes in Informatics, 2001.

[2] J. Cabot and R. Clarisó, "UML/OCL verification in practice," in *MoDELS'08*, Workshop on Challenges in MDE, 2008.

[3] A. D. Brucker and B. Wolff, "The hol-ocl book," 2010, http://www.brucker.ch/bibliography/download/2006/brucker.ea-hol-ocl-book-2006.pdf.

[4] J. Cabot, R. Clarisó, and D. Riera, "UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming," in *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE '07)*, pp. 547–548, ACM, 2007.

[5] M. Gogolla, J. Bohling, and M. Richters, "Validation of uml and ocl models by automatic snapshot generation," in *Proceedings of the 6th International Conference Unified Modeling Language*, pp. 265–279, Springer, 2003.

[6] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, pp. 266–290, 2002.

[7] J. Cabot, R. Clarisó, and D. Riera, "Verification of uml/ocl class diagrams using constraint programming," in *Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW '08)*, pp. 73–80, IEEE Computer Society, 2008.

[8] D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on UML class diagrams," *Artificial Intelligence*, vol. 168, no. 1-2, pp. 70–118, 2005.

[9] M. Balaban and A. Maraee, "A UML-based method for deciding finite satisfiability in description logics," in *DL'2008*, vol. 353 of *CEUR Workshop Proceedings*, 2008.

[10] A. D. Brucker and B. Wolff, "The HOL-OCL book," Tech. Rep. 525, ETH Zurich, 2006.

[11] A. Queralt and E. Teniente, "Reasoning on UML class diagrams with OCL constraints," in *ER'2006*, vol. 4215 of *LNCS*, pp. 497–512, Springer, 2006.

[12] A. Maraee and M. Balaban, "Eficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets," in *ECMDA-FA'2007*, vol. 4530 of *LNCS*, pp. 17–31, Springer, 2007.

[13] D. Jackson, *Software Abstractions: Logic, Language and Analysis*, MIT Press, 2006.

[14] J. Cabot, R. Clariso, E. Guerra, and J. de Lara, "Verification and validation of declarative model-to-model transformations through invariants," *Journal of Systems and Software*, vol. 83, no. 2, pp. 283–302, 2010.

[15] A. Shaikh, R. Clarisó, U. K. Wiil, and N. Memon, "Verification-driven slicing of uml/ocl models," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*, ACM, 2010.

[16] A. Shaikh, U. K. Wiil, and N. Memon, "Uost: Uml/ocl aggressive slicing technique for eficient verification of models," in *SAM*, pp. 173–192, 2010.

[17] A. D. Brucker and B. Wolff, "Hol-ocl: a formal proof environment for uml/ocl," in *Proceedings of the Fundamental Approaches to Software Engineering Conference (FASE '08)*, pp. 97–100, 2008.

[18] Object Management Group, "Unified modeling language," 2010, http://www.uml.org.

[19] Object Management Group, "Object constraint 6 language," 2010, http://www.omg.org/spec/OCL/2.0/.

[20] ArgoUML, Argouml, 2010, http://argouml.tigris.org.

[21] K. R. Apt and M. G. Wallace, *Constraint Logic Programming using ECL^{i}PS^e*, Cambridge University Press, 2007.

[22] B. Demuth, "The Dresden OCL toolkit and its role in Information Systems development," in *Proceedings of the 13th International Conference on Information Systems Development (ISD '04)*, Vilnius, Lithuania, 2004.

[23] "Uml2alloy," http://www.cs.bham.ac.uk/~bxb/UML2Alloy/.

[24] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL models in USE by automatic snapshot generation," *Software and Systems Modeling*, vol. 4, no. 4, pp. 386–398, 2005.

[25] MOVA, Mova, 2010, http://maude.sip.ucm.es/mova.

[26] M. Clavel, M. Egea, and V. T. D. Silva, "Mova: a tool for modeling, measuring and validating uml class diagrams," 2007.

[27] M. Clave, F. Durán, S. Eker et al., "Maude: specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002.

[28] D. J. R. J. A. Stafford and A. L. Wolf, "Chaining: a software architecture depen- dence analysis technique," Tech. Rep., University of Colorado Department of Computer Science, 1997.

[29] F. Lanubile and G. Visaggio, "Extracting reusable functions by flow graph-based program slicing," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 246–259, 1997.

[30] Q. Lu, F. Zhang, and J. Qian, "Program slicing: its improved algorithm and application in verification," *Journal of Computer Science and Technology*, vol. 3, no. 1, pp. 29–39, 1988.

[31] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[32] J. Cabot, R. Clariso, and D. Riera, "Papers and researchers: an example of an unsatisfiable uml/ocl model," http://gres.uoc.edu/UMLtoCSP/examples/Papers-Researchers.pdf.

[33] DBLP, Digital bibliography andy library project, http://guifre.lsi.upc.edu/DBLP.pdf.

[34] H. H. Kagdi, J. I. Maletic, and A. Sutton, "Context-free slicing of UML class models," in *Proceedings of the IEEE ICSM International Conference on Software Maintenance (ICSM '05)*, pp. 635–638, IEEE Computer Society, 2005.

[35] J.-D. Choi, J. H. Field, G. Ramalingam, and F. Tip, "Method and apparatus for slicing class hierarchies," http://www.patentstorm.us/patents/6179491.html.

[36] J. H. Bae, K. Lee, and H. S. Chae, "Modularization of the UML metamodel using model slicing," in *ITNG*, pp. 1253–1254, IEEE Computer Society, 2008.

[37] J. T. Lallchandani and R. Mall, "Slicing UML architectural models," in *ACM/SIGSOFT SEN*, vol. 33, pp. 1–9, 2008.

[38] J. Zhao, "Applying slicing technique to software architectures," CoRR, cs.SE/0105008, 2001.

[39] T. Kim, Y.-T. Song, L. Chung, and D. T. Huynh, "Dynamic software architecture slicing," in *Proceedings of the 23rd International Computer Software and Applications Conference (COMPSAC '99)*, pp. 61–66, IEEE Computer Society, 1999.