Verification-Driven Slicing of UML/OCL Models

Asadullah Shaikh
Universitat Oberta de Catalunya Barcelona, Spain
ashaikh@uoc.edu

Robert Clarisó
Universitat Oberta de Catalunya Barcelona, Spain
rclariso@uoc.edu

Uffe Kock Wiil
The Maersk-McKinney Moller Institute
University of Southern Denmark Odense, Denmark
ukwiil@mmmi.sdu.dk

Nasrullah Memon
The Maersk-McKinney Moller Institute
University of Southern Denmark Odense, Denmark
memon@mmmi.sdu.dk

ABSTRACT

Model defects are a significant concern in the Model-Driven Development (MDD) paradigm, as model transformations and code generation may propagate errors to other notations where they are harder to detect and trace. Formal verification techniques can check the correctness of a model, but their high computational complexity can limit their scalability. In this paper, we consider a specific static model (UML class diagrams annotated with unrestricted OCL constraints) and a specific property to verify (satisfiability, i.e., "is it possible to create objects without violating any constraint?"). Current approaches to this problem have an exponential worst-case runtime. We propose a technique to improve their scalability by partitioning 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.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.2 [**Programming Languages**]: Language Classifications; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Verification

Keywords

MDD, UML, OCL, Model Slicing, Formal Verification

1. INTRODUCTION

Model-Driven Development (MDD) advocates for a shift in the software engineering process: using models, instead of code, as the primary development artifact. Within MDD, tool support for model design, analysis and transformation is essential and scalability is an important requirement.

There are formal verification tools for automatically checking correctness properties on models, but the lack of scalability is usually a drawback and its improvement will be the goal of this paper. We will consider the analysis of the static elements of a software specification, modeled as a UML class diagram. Complex integrity constraints will be expressed in the Object Constraint Language (OCL). In this context, the fundamental correctness property of a model is *satisfiability* [1,4,19]: "is it possible to instantiate the model without violating any integrity constraint?".

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, a pair of constraints C_1 and C_2 are not redundant if the following is satisfiable: $(C_1 \wedge^{\neg} C_2) \vee (^{\neg}C_1 \wedge C_2)$, i.e., it is possible to satisfy C_1 but not C_2 or vice versa.

Reasoning on UML class diagrams without OCL integrity constraints is an EXPTIME-complete problem [2]. Furthermore, the addition of unrestricted[1] OCL constraints makes the problem undecidable. Current solutions for checking satisfiability employ formalisms such as description logics [1], higher-order logics [3], database deduction systems [19], linear programming [17], SAT [11] or constraint satisfaction problems [2,4]. 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.

A review of sample UML/OCL models highlights two observations which are relevant to this problem. First, models

[1]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.

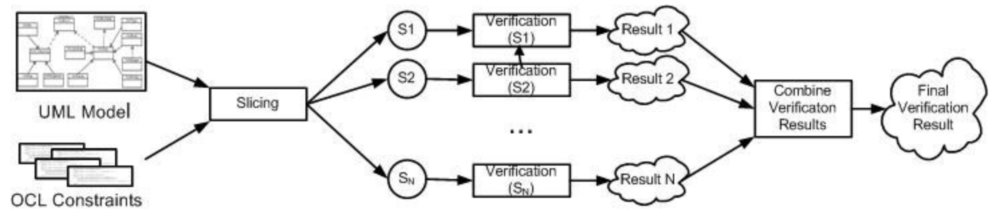


Figure 1: High-level description of the slicing process.

typically contain elements which are either unconstrained or constrained in a trivially satisfiable way. For instance, attributes acting as identifiers should have unique values, but often there are no other constraints on these attributes. A second observation is that some constraints refer to independent entities. For example, constraints about the password of a user and the price of a product are likely to be unrelated.

These observations can be used to improve the scalability of verification methods for satisfiability. Our proposal is based on *model slicing*: given an input UML/OCL model, the diagram and its constraints will be automatically partitioned into submodels while abstracting unnecessary model elements. The structure of the class diagram (associations and class hierarchies) and the OCL invariants (abstract syntax tree) will guide the partitioning process. Intuitively, the underlying idea is that all constraints restricting the same model element should be verified together and therefore belong to the same slice. Then, satisfiability of each slice is checked independently and the results are combined to assess the satisfiability of the entire model. Figure 1 illustrates the overall flow. To ensure soundness, slicing should not alter the outcome of the verification.

The remainder of the paper focuses on the definition of the slicing algorithm and the evaluation of its benefits. Section 2 introduces a running example and a more detailed overview of the slicing procedure. Section 3 focuses on the analysis of OCL constraints while Section 4 explores the structural analysis of the class diagram. Section 5 describes experimental results measuring how slicing reduces verification time. Section 6 shows a larger real world case study where the benefits of slicing are illustrated. In Section 7, related work is presented. Finally, Section 8 provides the conclusions and identifies directions of future work.

2. OVERVIEW OF UML/OCL SLICING

The approach: The input of our method is a UML class diagram annotated with OCL invariants. As an example, Figure 2 introduces class diagram that will be used throughout the paper, modeling the information system of a bus company. Several integrity constraints are defined as OCL invariants.

Our goal is to determine whether the input class diagram has legal instances, that is, instances that satisfy all integrity constraints. An instance of a UML class diagram is a collection of objects (according to the class definitions) and a collection of links between them (according to the associations). The output of the tool will either be "satisfiable" or "unsatisfiable". In case of satisfiability, a sample instance proving the satisfiability will be computed as well.

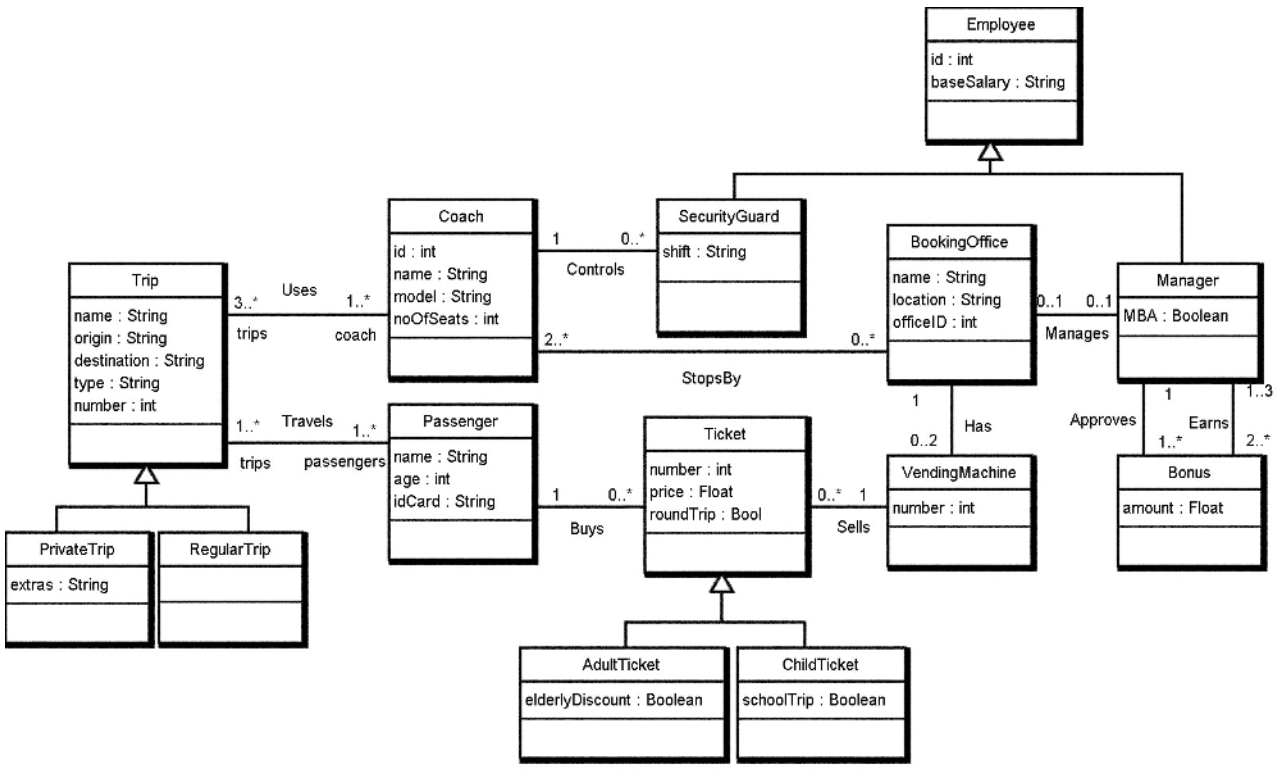
Two different notions of satisfiability will be considered for verification: *strong* and *weak* satisfiability [1, 4, 19]. A class diagram is weakly satisfiable if it is possible to create a legal instance which is non-empty, i.e., it contains at least one object from *some* class. On the other hand, 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. Some parts of the slicing algorithm will work differently depending on the satisfiability notion being verified.

The algorithm works by partitioning the UML/OCL class diagram into a set of disjoint *slices*. A slice S of a UML/OCL class diagram D is a subset of the original model: another valid UML/OCL class diagram where any element (class, association, inheritance, aggregation, invariant, . . .) appearing in S also appears in D, but the reverse does not necessarily hold. Figure 3 represents the slices for strong satisfiability. Each slice is verified independently and the verification result of the whole model is obtained by combining the results of all slices. If we are checking strong satisfiability, it is necessary to check whether *all* slices are strongly satisfiable. On the contrary, if we are checking weak satisfiability, it is sufficient to ensure that *at least one* slice is weakly satisfiable.

Preserving satisfiability: The fundamental requirement of this procedure is that it should preserve the outcome of the verification: the answer provided by the verification with slicing should be the same as the one given by a verification tool without slicing.

Each slice is a disjoint subset of the integrity constraints and a disjoint fragment of the original class diagram. As each slice is less constrained than the original model, it is clear that if the original model was satisfiable, the slices will also be satisfiable. Therefore, it is only necessary to ensure that if the original model was unsatisfiable, the answer will also be "unsatisfiable": if we are checking strong satisfiability, at least one slice will be strongly unsatisfiable, and if we are checking weak satisfiability, all slices will be weakly unsatisfiable. A formal proof is out of the scope of this paper but we will try to provide some insight on how satisfiability is preserved.

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 or 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, e.g., self.x = 7. Finally, it is possible that the unsatisfiability arises from a combination of both factors, e.g., the values of



context Coach **inv** MinCoachSize: self.noOfSeats $\geq$ 10
context Coach **inv** MaxCoachSize: self.trips −>forAll(t | t.passengers −>size() $\leq$ noOfSeats)
context Trip **inv** CorrectTripDestination: **not** self.origin = self.destination
context Ticket **inv** UniqueTicketNumber: Ticket::allInstances() −>isUnique (t | t.number)
context Ticket **inv** MachineNumber: self.name=self.vendingMachine.bookingOffice.location.concat(self.number.toString())
context Passenger **inv** NonNegativeAge: self.age $\geq$ 0

Figure 2: UML/OCL class diagram used as running example (model Coach).

some attributes require a certain number of objects to be created which contradict 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, i.e., the constraints can be satisfied on their own but not simultaneously. To ensure that unsatisfiability is propagated into the slices, three conditions should be guaranteed:

1. No potentially unsatisfiable constraint should be removed from the problem.
2. If there are two or more constraints whose interaction could be unsatisfiable, none of them should be removed from the problem.
3. All constraints referring to the same model element should appear together in the same slice, i.e., their interaction should not be split into different slices.

In order to ensure that conditions (1-3) hold, the UML/OCL model has to be analyzed before slicing. The analysis should reveal which parts of the model can be abstracted or partitioned safely. The following sections focus on this analysis at two levels: UML and OCL. In Section 3, a traversal of the syntax tree of each OCL constraint identifies which classes, attributes and navigations are being restricted. Additional analysis identifies trivial constraints and constraints that can be checked independently. In Section 4, dependencies among the number of objects in each class, like inheritance hierarchies or multiplicity constraints, are studied.

3. ANALYSIS OF OCL CONSTRAINTS

OCL allows the definition of *expressions* on UML models. An expression which evaluates to true or false, e.g., a class invariant, will be called a *constraint*. OCL can also be used to define the result of *query operations*, which can then be invoked inside other expressions.

Invariant	Support	Attributes	Navigations
MinCoachSize	Coach	Coach.noOfSeats	None
MaxCoachSize	Coach, Trip, Passenger	Coach.noOfSeats	Travels, Uses
CorrectTripDestination	Trip	Trip.(origin,destination)	None
MachineNumber	VendingMachine, BookingOffice, Ticket	Ticket(name,number) BO.location	Sells, Has
UniqueTicketNumber	Ticket	Ticket.number	None
NonNegativeAge	Passenger	Passenger.age	None

Table 1: Support, attributes and navigations in the running example.

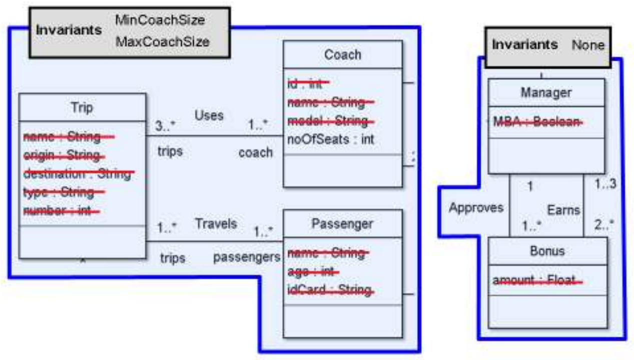


Figure 3: Slices for the verification of strong satisfiability in the running example.

Any OCL expression is defined within the *context* of a type. Typically, an OCL expression involves several objects from one or more classes of the model. To get a starting object, we can use the keyword *self*, which denotes an object of the context type; or the method allInstances() which can be used to access all objects of a given type, e.g., Trip::allInstances() returns a set of all objects of class Trip. Given an object, OCL provides operators to read the values of its attributes (*attribute access*) and access the objects connected to it through associations (*navigation*). Combining these operators with arithmetic, logic and relational operators, iterators and user-defined query operations, it is possible to write complex constraints about UML models.

This section describes how to analyze OCL invariants in order to extract information relevant to its satisfiability. We are interested in identifying which model elements are constrained by an invariant, as interactions between constraints appear when two or more constraints restrict the same model elements.

3.1 Constraint Support

The *support* of an OCL expression is the subset of classes of the class diagram referenced by the expression. If the expression is a query operation, it contains the types whose objects are explored to evaluate the query. For invariants, the support describes the set of classes restricted by the constraint. This information will be used identify classes that appear together in the same constraint and therefore must be analyzed within the same slice. Formally, the support of an expression E contains the following types:

1. The context type where E is defined and all its supertypes, as long as the "self" variable appears within E.
2. The type of each association end navigated within E.
3. Each type referenced explicitly in E by the operation Type::allInstances() or by a type check or conversion operation, e.g., oclIsKindOf, oclIsTypeOf or oclAsType.
4. The union of the supports of all query operations invoked from E.

Another piece of information required by the remaining analysis steps is the set of attributes and navigations used in each invariant. This information can be gathered with a straightforward traversal of the OCL syntax tree. Table 1 summarizes all this data for the invariants of the running example.

3.2 Local and Global Constraints

Some parts of a verification problem can be checked in isolation within the boundaries of a class and without affecting the big picture. Intuitively, if there is a constraint on an attribute which is not used anywhere else in the model, we can split the verification problem in two separate subproblems: checking that the constraint on the attribute is feasible and verifying the rest of the system. This section will present the tools to identify these local constraints.

An expression 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. 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*.

In other words, a local expression can be defined as follows: (a) it does not use navigations through associations and, (b) it does not call allInstances(), (c) it does not use attributes defined in a superclass, (d) it does not call any global query operation and (e) it does not perform any type check or type conversion operation. Table 2 shows some examples of local and global expressions written in the context of class *Trip*.

Some attributes may appear in local constraints, global constraints or both. We are interested in detecting those attributes that can be studied locally, like those that do not appear in global constraints and are not related to attributes that appear there. In this sense, the set of *global* attributes will be iteratively defined as follows: (a) the attributes used in global expressions plus (b) the attributes used in local expressions where there is at least one global attribute. All other attributes of the model will be called *local*. A local expression which uses only local attributes will be called *strongly local*.

It should be noted that according to our definition, the result of a strongly local invariant does not depend on (a) attributes outside those mentioned in the expression or (b) the

Type	Expression (context Trip)	Description
Local	self.origin $\neq$ self.destination	Attribute access
Global	**not** self.passengers−>isEmpty()	Navigation
Global	Ticket::allInstances()−>isUnique(t|t.number)	allInstances()
Global	self.oclIsTypeOf("PrivateTrip")	oclIsTypeOf()

Table 2: Examples of local and global invariants.

Pattern	Condition
Type::allInstances() −>isUnique(at)	Key constraint if attribute is not constrained anywhere else.
self.at op exp	Derived value constraint if attribute is not used anywhere else and expression does not involve attribute.
A or B	Trivially satisfiable if either A or B are satisfiable.
A and B	Trivially satisfiable if either A and B are satisfiable.
A implies B $=\neg$A$\vee$B	Trivially satisfiable if either $\neg A$ or B are satisfiable.
Not A	Trivially satisfiable if A is trivially satisfiable and it is not a key constraint.
self.navigation−>isUnique(at)	Trivially satisfiable if attribute is not used anywhere else.

Table 3: Patterns with Conditions

number of objects in any class. The only chance of potential interaction with other invariants is with other strongly local invariants of the same class, if they have any attribute in common. Therefore, strongly local invariants of a class can be analyzed separately from the rest of the model. The division into subproblems is the following:

- A problem defined by the class, its local attributes and its strongly local invariants (which can be further partitioned if these invariants restrict disjoint sets of attributes).

- Another problem defined by the original model, removing the attributes and constraints that appear in the first subproblem.

In our running example, invariants MinCoachSize, NonNegativeAge and CorrectTripDestination are all local invariants. Of these, invariant MinCoachSize is not strongly local as the attribute "noOfSeats" is also used in the global invariant MaxCoachSize. The remaining invariants, NonNegativeAge and CorrectTripDestination can be abstracted from the model together with the attributes they reference and their satisfiability can be checked independently.

3.3 Trivially Satisfiable Constraints

A final analysis that can improve the efficiency of satisfiability verification is the detection and removal of trivially satisfiable invariants from the UML/OCL class diagram. Detecting satisfiable constraints is as hard as satisfiability itself, so we restrict ourselves to consider typical patterns which may arise in different applications.

The first trivially satisfiable pattern which can be safely removed is the *key constraint*, stating that a given attribute must be unique, e.g., Type::allInstances() −>isUnique(obj | obj.attr). If the attribute is of type Integer, Float or String and it is not referenced by any other constraint, it can be trivially satisfied: a different value can be assigned to each potential instance, e.g., 1, 2, 3, ... The verification engine does not need to spend time computing the value of the attribute in each object and enforcing uniqueness among different objects.

Another trivially satisfiable pattern which can also be removed is the *derived value constraint*, where the value of one attribute depends on the values of other attributes. The pattern 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*. By "safe" we mean 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().

Intuitively, this constraint cannot make the model unsatisfiable: if an instance for the rest of the model can be created, it is simply a matter of evaluating *expression* to find out the right value of *attrib*. The conditions on *expression* (no self-references, no undefined values) guarantee that the evaluation always computes a feasible value for *attrib*. Table 3 briefly summarizes the patterns and conditions where the column *Pattern* shows the possible expressions and the column *Condition* illustrates the criteria of the given patterns.

Regarding the feed-back provided to the user, it is possible to hide the fact that these constraints have been abstracted. If the verification tool provides an instance of the model as an answer, a post-processing phase can add the abstracted attributes and assign them correct values according to the constraints.

Considering the running example, invariant MinCoachSize is a derived value constraint where the expression is the constant 0. However, this invariant is not trivially satisfiable and therefore cannot be abstracted, because the attribute "noOfSeats" is also constrained by the invariant MaxCoachSize. On the other hand, constraints NonNegativeAge, CorrectTripDestination and MachineNumber are derived value constraints which can be abstracted. Finally, invariant UniqueTicketNumber is a key constraint which can also be abstracted.

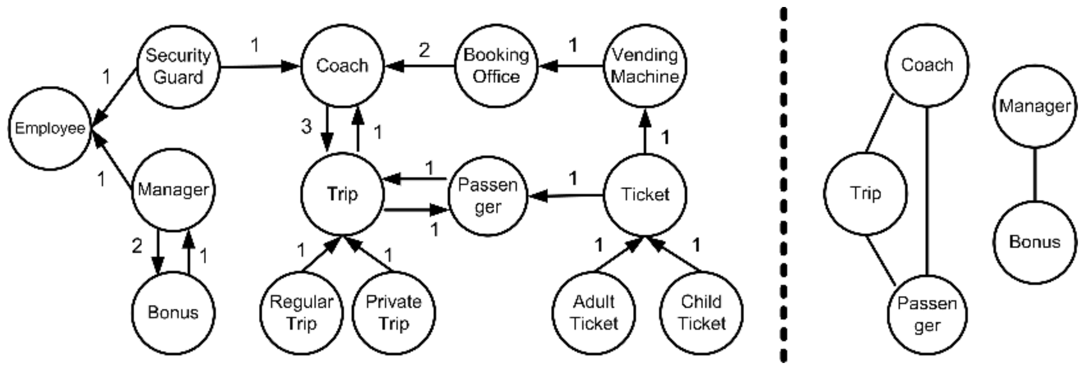


Figure 4: Flow graph (left) and dependency graph (right) for the running example.

4. ANALYSIS OF UML CLASS DIAGRAMS

For the sake of brevity and without loss of generality, in this section we will consider a UML class diagram composed of binary associations and inheritance relations. The remaining features of class diagrams like associative classes or n-ary associations can be expressed in terms of binary associations (and potentially additional OCL constraints) [9].

In this phase, we will compute a graph-based representation (*dependency graph*) that captures the dependencies of the elements within the UML/OCL class diagram. Then, the computation of slices will simply consist of computing the *connected components* of the graph, i.e., the maximal subgraphs where there is a path among each pair of vertices. Intuitively, each connected component represents a set of interdependent constraints which have to be analyzed as a whole.

A dependency graph is an undirected graph where each vertex is a class of the model. The core challenge is the definition of the conditions under which two vertices will be connected: they should be as aggressive as possible (removing irrelevant dependencies) while being conservative (related vertices will not be separated under any circumstance).

In order to define these relationships, we will use an auxiliary graph-based representation called *flow graph*. A flow graph is a labeled directed pseudograph, i.e., there can be arcs from one vertex to itself and multiple arcs between two vertices. The vertices of the flow graph are the classes of the class diagram and the labels in the arcs are non-negative integers. An arc $X \overset{n}{\to} Y$ has the meaning "if there is an object in class X, at least n objects of class Y must exist". Using this definition, there is an arrow $X \overset{n}{\to} Y$ if:

- X is a subclass of Y ($n = 1$): Each object of a subclass is also an object of the superclass.
- There is an association between X and Y and the lower bound of the multiplicity of the association end in Y is n.

Arcs with a label 0 can be removed because they are not imposing any constraint. Multiple arcs between two vertices can be replaced by a single arc labeled with maximum label among them. For example, Figure 4 illustrates the flow graph for the running example after these simplifications.

Intuitively, a path in the flow graph among vertices X and Y establishes a dependency from X to Y. A cycle defines a cyclic dependency and it is therefore a possible source of unsatisfiability. Any cycle where the maximum label is 1 is inherently satisfiable, and it will be called *safe*. But cycles where (a) the maximum label ≥ 2 and (b) there are two or more participating associations/inheritances relations which also form a cycle in the class diagram *can* be unsatisfiable. Such cycles will be called *unsafe*. In our running example (Figure 4), there are three cycles: Trip-Coach, Trip-Passenger and Manager-Bonus. The first two are safe (they only involve one association so there is no cycle in the class diagram) while the third one is unsafe (two associations participate in the cycle and there is a multiplicity with lower bound 2).

Using this information, the dependency graph will be created in two steps. In the first step, we identify classes which are *potentially unsatisfiable*, i.e., classes constrained by OCL invariants and classes belonging to an unsafe cycle:

1. Create a vertex for each class that appears in the constraint support of an OCL constraint.
2. Add an edge $X - Y$ if both X and Y belong to the constraint support of the same constraint.
3. Create a vertex (if it does not previously exist) for each class that appears in an unsafe cycle in the flow graph.
4. Add an edge $X - Y$ among all vertices participating in the same unsafe cycle.

In the second step, we iteratively add classes that constrain vertices already in the dependency graph. Let X and Y be a pair of vertices in the dependency graph, where X and Y can be the same vertex, and Z a class that does not appear in the dependency graph. Then, if there is a path from X to Z and from Z to Y *in the flow graph*, vertex Z must added to the dependency graph together with edges $X - Z$ and $Y - Z$. This process propagates dependencies between potentially unsatisfiable classes that cross through other classes. In our running example, the resulting flow graph is shown in Figure 4, with two connected components: one coming from the unsafe cycle in the flow graph *Manager-Bonus* and another coming from the constraints *Min/Max-CoachSize*, formed by classes *Coach*, *Trip* and *Passenger*.

From the dependency graph, it is possible to extract its connected components. Each component defines a slice of the class diagram that can be analyzed independently: the set of classes from the class diagram, the set of associations and inheritance hierarchies among them, the invariants that have some of these classes in their support and the attributes referenced by any of those invariants. For example, Figure 3 highlights the final slices passed to the verification tool for strong satisfiability. Strikethrough text indicates attributes from the original model which have been abstracted in the slice. Notice how thanks to the detection of trivially satisfiable invariants described in the previous Section, some attributes like *origin* which were originally constrained by an invariant can be simply abstracted.

With this approach, the slices of the class diagram correspond to those fragments that could be unsatisfiable. The implication is "if the slices can be populated, then the remaining classes can be populated as well". But what happens if these slices cannot be populated? This does not matter for strong satisfiability, as *all* classes must be populated so any failure means the whole model is unsatisfiable. However, in weak satisfiability it could be the case that all slices are unsatisfiable but some of the remaining classes can be satisfied independently. Considering our running example, let us consider class *Employee*: creating an employee does not impose any obligation on any other class of the model. Thus, it is clear that this class can be populated and the model is weakly satisfiable. Formally, if there is any class X such (a) X does not appear in the dependency graph and (b) the flow graph has no path from X to a class in the dependency graph, the model is weakly satisfiable. In this case X and any classes which depend from X can be populated even if no class if the dependency graph can be populated. In our running example, class *Employee* is the only class which exhibits this trait.

5. EXPERIMENTAL RESULTS

In this section, we attempt to quantify the speed-up achieved by slicing. To this end, we have developed a prototype implementation of the slicing procedure on top of the tool UMLtoCSP [4]. UMLtoCSP transforms verification problems on UML/OCL class diagrams into *constraint satisfaction problems* (CSP) which can be solved by a constraint solver. Solutions to the CSP are instances of the diagram which prove or disprove the property being verified.

We compare the verification time of several UML/OCL class diagrams using (1) the original tool UMLtoCSP and (2) the tool UMLtoCSP with slicing. In each example, the property being verified is strong satisfiability. Table 4 describes the set of benchmarks used for our comparison: the number of classes, associations, invariants and attributes. For each class diagram, we also indicate whether it is strongly satisfiable or not. The benchmarks "Company", "Script" and "Cycle" were programmatically generated, in order to test large input models. Of these models, we consider the "Script" models to be the best possible scenario for slicing (large models with many attributes and very few constraints). The models "Paper-Researcher", "Atom-Molecule", "Company" and "Cycle" serve as worst-case scenarios (models with many interdependent constraints, designed so they cannot be sliced).

UMLtoCSP has a set of parameters that can have a strong influence on its runtime. These parameters set an upper bound on the size of the instance (number of objects per class, number of links per association) and the domain of attributes (set of feasible values for each attribute). In UMLtoCSP, verification is not *complete* in the sense that it will only explore potential instances within these bounds. Nevertheless, the size of the solution space to be explored by UMLtoCSP is exponential in terms of these parameters. Therefore, large values of the parameters will make the comparison more favorable towards slicing, as abstracting attributes and classes will cause a larger reduction of the solution space. In our analysis, we have considered small but reasonable values for parameters: at most 4 objects will be created for each class, at most 10 links for each association and each attribute will have at most 10 distinct values.

Table 5 shows the experimental results computed using a Intel Core 2 Duo Processor 2.1Ghz with 2Gb of RAM. All times are measured in seconds and a time-out limit has been set at 2 hours (7200 seconds). For each example, we describe the original verification time (OVT), the number of slices in which the model is divided, the number of attributes that we manage to abstract, the time required to perform all the UML/OCL slicing analysis (ST) and the verification time after the slicing (SVT).

The first conclusion is that slicing is a very fast procedure even in diagrams with hundreds of classes. As expected, the effectiveness of the technique depends on the specific model being analyzed: small models and models where UMLtoCSP already performed well gain little from slicing. This also happens with models where there are no unconstrained attributes and all classes and constraints are interdependent. In the worst case, the verification time with slicing is the same as that without slicing. But in models where slicing manages to partition the model and abstract attributes, the speed-up reaches several orders of magnitude. Therefore, its success will depend on the type of models where it is applied. Small models which have been manually preprocessed for verification will gain little from slicing. However, models created for other purposes or models generated through automatic transformation can benefit greatly from the application of slicing. 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.

6. SLICING ALLOY SPECIFICATION

In this section, we have applied our slicing technique to the DBLP (Digital Bibliography and Library Project) structural schema programmed in the Alloy specification. The schema of the DBLP system is modeled a UML class diagram [6]. It is a computer science bibliographical website, and has existed since the 1980's. 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 26 integrity constraints. It is divided into two types of integrity constraints: identification and other 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.

The approach is manually implemented over the DBLP in order to show how fast it generates satisfying instances of the example before and after the slicing is applied. The same model is taken for slicing in the Alloy [10] to check the advantages of slicing. The execution time is largely dependent on the defined scope, therefore, in order to analyze

Example	Classes	Associations	Attributes	Invariants	Satisfiable?
Atom-Molecule	2	1	6	1	Yes
Paper-Researcher	2	2	5	4	No
Coach	15	10	27	6	Yes
Production System	50	30	72	5	Yes
Company	100	100	100	100	Yes
Script 1	100	53	122	2	Yes
Script 2	500	227	522	5	Yes
Script 3	1000	505	1022	5	Yes
Cycle 1	10	10	10	10	No
Cycle 2	100	100	100	100	No

Table 4: Description of the UML/OCL benchmarks.

Example	OVT	Slices	Attr	ST	SVT
Atom-Molecule	0.03s	1	3	0.00s	0.03s
Paper-Researcher	0.04s	1	0	0.00s	0.04s
Coach	5008.76s	2	26	0.00s	0.18s
Production System	3605.35s	4	59	0.02s	0.03s
Company	0.08s	1	0	0.00s	0.08s
Script 1	Time-out	2	117	0.02s	0.03s
Script 2	Time-out	4	509	0.09s	0.02s
Script 3	Time-out	4	1009	0.29s	0.34s
Cycle 1	Time-out	1	10	0.00s	Time-out
Cycle 2	Time-out	1	100	0.00s	Time-out

OVT	Original Verification Time	**Attr**	# of abstracted attributes
SVT	Total verification time for all slices	**ST**	Slicing Time

Table 5: Description of experimental results.

the efficiency of verification, scope 7 is limited. The Alloy will examine the entire DBLP model with up to 7 objects, and try to find one that violates the property. For example, saying scope 7 means that Alloy will check the model whose top level signatures have up to 7 instances. 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 6 summarizes the experimental results obtained using the Alloy analyzer before and after slicing, running on a Intel Core 2 Duo Processor 2.1Ghz with 2Gb of RAM. All times are measured in milliseconds (ms). 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, each scope, after slicing time is also defined which is the sliced translation time (STT), sliced solving time (SST) and the summation of STT and SST, which is equivalent to the summation of TT and ST. The only difference is that the total execution time varies before and after slicing. Similarly, the column speed up shows the efficiency obtained after the implementation of the slicing approach. The speedup is achieved using the equation below:

$$[1 - \{(STT + SST)/(TT + ST)\}] * 100$$

Previously, it took 1453 ms (scope 7) for the execution of the DBLP. Using the approach for the slice computed by this method, it takes only 828 ms (scope 7) to generate a satisfying instance for the slice. It is an improvement of 43% which is a marked progress in total execution time. In addition, the improvement can also be achieved for larger scopes as well. For instance, the results up to the scope of 35 can be achieved for the DBLP structural schema. However, without slicing we could run the analysis up to the scope of 19.

7. RELATED WORK

Slicing techniques can be classified according to two criteria: the *entity* being sliced (e.g., a program, a UML model, an ontology, . . .) and the *goal* of the slicing process (e.g., synthesis, analysis, optimization, visualization, comprehension, . . .). Intuitively, all slicing techniques proceed in two steps: first, the subset of elements of interest that should appear in the slice is identified; second, elements which depend on elements of the slice are iteratively added to the slice. The notions of “element”, “element of interest” and “dependency between elements” are completely determined by *what* is being sliced and *why*.

Program slicing [23, 25] techniques work at the level of source code. Given a set of variables of interest and a program location which are provided as input, program slicing computes the set of statements of the program that can affect (backward) or be affected (forward) by those variables. The applications of program slicing include program analysis, optimization, verification and comprehension. Slicing

Before Slicing				After Slicing			
Scope	TT	ST	TT+ST	STT	SST	STT+SST	Speedup %
2	125ms	47ms.	172ms	110ms	31ms	141ms	18%
3	187ms	78ms	265ms	125ms	62ms	187ms	29%
4	281ms	172ms	453ms	219ms	78ms	297ms	34%
5	473ms	190ms	663ms	299ms	110ms	409ms	38%
6	671ms	344ms	1015ms	438ms	156ms	594ms	41%
7	969ms	484ms	1453ms	672ms	156ms	828ms	43%

TT	Translation Time	**ST**	Solving Time
STT	Sliced Translation Time	**SST**	Sliced solving Time

Table 6: Description of experimental results (Alloy).

has also been used in the analysis of *architectural specifications* of a software system [13, 21]. In this context, extracting the set of components related to a component of interest can facilitate component reuse and provide a high-level view of the architecture that helps in its comprehension. Another type of program slicing is used for *Declarative Specifications* [24]. This work proposes a tool known as *Kato* which relies on heuristics to identify "core" (slices) and it is targeted towards the relational logic underlying Alloy. Few details are provided on the set of heuristics being used.

Ontologies provide a formal description of a set of concepts and their relationships. General purpose ontologies may represent a large number of concepts and their size makes them impractical for many applications. Several approaches [5,18,22] focus on pruning large ontologies to produce smaller ontologies which are more manageable. In this case, concepts of interest are identified by the modeler and provided as the input of the slicing method.

Slicing methods have also been proposed the management of different types of UML models. *Context-free slicing* [12] provides a framework for defining model slices in UML diagrams, e.g., class diagrams. This work proposes a general theory of model slicing which has to be adapted to each specific goal by defining a slicing criterion suitable for our goal. There is no discussion on the definition of suitable slicing criteria for verification. A different approach focusing in class diagram comprehension is the use of coupling metrics [14] to slice large models for visualization. This type of approach would not be well-suited for verification purposes, as metrics do not provide guarantees about the properties satisfied by the partitions. Finally, the slicing of models consisting of both UML class diagrams and UML sequence diagrams is considered in [15]. A common representation, called Model Dependency Graphs, is used to encode both types of diagrams. Again, the slicing criterion must be provided as an input to the algorithm.

In contrast to these previous works, this paper describes a slicing criterion oriented towards the verification of satisfiability of UML/OCL class diagrams. Previous works either do not target UML class diagrams or do not consider OCL (other than as a notation to express slicing criteria) and none propose a slicing criterion for verification.

Another source of relevant work appears in the underlying theorem provers and solvers used to check satisfiability in UML/OCL models. At this level, similar concepts for partitioning, symmetry-breaking and other optimizations have been considered extensively, for instance [7,16,24]. We claim that slicing *before* the translation into a formalism like SAT or CSP is worthwhile due to several reasons. First, slicing analysis is independent of the underlying formalism, so it can benefit a variety of tools. Furthermore, at this level of abstraction the problem is smaller, so it is feasible to perform more complex analysis. Finally, we can take advantage of our knowledge of the semantics of UML/OCL and the property being verified, information which can be lost in the translation into the formalism. For instance, the removal of derived value constraints proposed in Section 3.3 would not be possible without precise information about the property being checked.

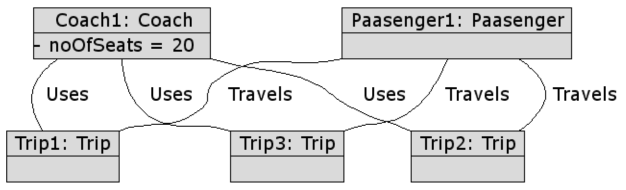
8. CONCLUSIONS AND FUTURE WORK

This paper presents an innovative slicing technique for UML/OCL class diagrams aimed at making the verification of satisfiability more efficient. The approach receives as input a UML class diagram annotated with OCL constraints and automatically breaks it into submodels whose satisfiability can be analyzed independently. Then, the satisfiability of the original model can be established by checking if at least one submodel (weak satisfiable) or all submodels (strong satisfiable) are satisfiable. A benefit of this approach is that it is independent of the underlying formalism used to check satisfiability and can therefore be applied in many existing tools.

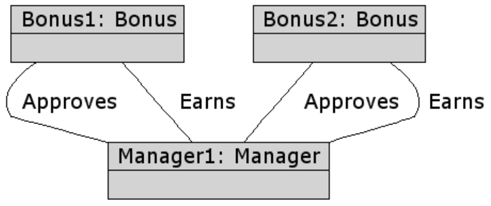
A prototype implementation of the slicing procedure has been developed on top of the tool UMLtoCSP. The object diagram of running example is shown in Figure 5. Experimental results show that slicing can produce a significant speed-up in the verification time. The amount of speed-up achieved by this method depends on the specific model, from none to several orders of magnitude.

As the overhead introduced by slicing analysis is negligible, we claim that slicing is a useful addition to any UML/OCL satisfiability checking toolkit. Furthermore, we demonstrated slicing technique on a real world case study named as DBLP conceptual to analyze the benefits. This real world case study is programmed in Alloy which is a famous tool and widely used for verification of models. We applied slicing and achieved drastic speed-up in another tool as well.

As our future work, we plan to explore two research directions. First, we plan to investigate more aggressive slicing criteria which can still preserve the satisfiability of the model after partitioning. The approach presented in this paper is very conservative in several ways, for example it only considers disjoint slices. Relaxing the proposed approach (while ensuring the preservation of satisfiability) would pro-



(a) Submodel 1 of 'Model Coach'



(b) Submodel 2 of 'Model Coach'

Figure 5: UMLtoCSP Output of Model Coach

vide more opportunities for slicing a class diagram. Second, the partition of a model into submodels can provide a useful feedback in case of unsatisfiability: if a model is unsatisfiable, it is possible to identify in which submodel(s) that happens. Designers can therefore focus their attention in the incorrect submodels while ignoring the rest of the model. This feedback is coarser than those approaches designed to compute a "minimum set of conflicting constraints", e.g., unsat cores in Alloy or [20], but it is computed with very little overhead and it can still be useful to designers.

Acknowledgements

This work has been partly supported by the Ministerio de Ciencia y Tecnologia under project TIN2008-00444 and by an UOC-IN3 Doctoral Grant. We would like to thank Jordi Cabot and Jordi Conesa for their useful comments.

9. REFERENCES

[1] M. Balaban and A. Maraee. A UML-based method for deciding finite satisfiability in Description Logics. In *DL'2008*, volume 353 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[2] D. Berardi, D. Calvanese, and G. D. Giacomo. Reasoning on UML class diagrams. *AIntelligence*, 168:70–118, 2005.

[3] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[4] J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE'2007*, pages 547–548. ACM, 2007.

[5] J. Conesa and A. Olivé. Pruning ontologies in the development of conceptual schemas of information systems. In *ER'2004*, volume 3288 of *LNCS*, pages 122–135. Springer, 2004.

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

[7] V. Durairaj and P. Kalla. Guiding CNF-SAT search via efficient constraint partitioning. In *ICCAD'04*, pages 498–501. IEEE Computer Society, 2004.

[8] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.

[9] M. Gogolla and M. Richters. Expressing UML Class Diagrams Properties with OCL. In *AOM with the OCL*, volume 2263 of *LNCS*, pages 86–115. Springer, 2001.

[10] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

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

[12] H. H. Kagdi, J. I. Maletic, and A. Sutton. Context-free slicing of UML class models. In *ICSM'05*, pages 635–638. IEEE Computer Society, 2005.

[13] T. H. Kim, Y. T. Song, L. Chung, and D. Huynh. Software architecture analysis: A dynamic slicing approach. *International Journal of Computer & Information Science*, 1(2):91–103, 2000.

[14] R. Kollmann and M. Gogolla. Metric-based selective representation of uml diagrams. In *CSMR'02*, pages 89–98. IEEE Computer Society, 2002.

[15] J. T. Lallchandani and R. Mall. Slicing UML architectural models. In *ACM / SIGSOFT SEN*, volume 33, pages 1–9, 2008.

[16] Y. C. Law and J. H. Lee. Symmetry breaking constraints for value symmetries in constraint satisfaction. *Constraints*, 11(2-3):221–267, 2006.

[17] A. Maraee and M. Balaban. Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In *ECMDA-FA'2007*, volume 4530 of *LNCS*, pages 17–31. Springer, 2007.

[18] B. J. Peterson, W. A. Andersen, and J. Engel. Knowledge bus: Generating application-focused databases from large ontologies. In *KRDB '98*, volume 10 of *CEUR Workshop Proceedings*, pages 2.1–2.10. CEUR-WS.org, 1998.

[19] A. Queralt and E. Teniente. Reasoning on UML class diagrams with OCL constraints. In *ER'2006*, volume 4215 of *LNCS*, pages 497–512. Springer-Verlag, 2006.

[20] G. Rull, C. Farré, E. Teniente, and T. Urpí. Computing explanations for unlively queries in databases. In *CIKM'07*, pages 955–958. ACM, 2007.

[21] J. A. Stafford, D. J. Richardson, and A. L. Wolf. Architecture-level dependence analysis in support of software maintenance. In *ISAW'98*, pages 129–132, 1998.

[22] B. Swartout, P. Ramesh, K. Knight, and T. Russ. Toward distributed use of large-scale ontologies. *AAAI Symp. on Ontological Engineering*, pages 138–148, 1997.

[23] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[24] E. Uzuncaova and S. Khurshid. Kato: A program slicing tool for declarative specifications. In *ICSE '07*, pages 767–770. IEEE Computer Society, 2007.

[25] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.