Efficient Verification-Driven Slicing of UML/OCL Class Diagrams

Asadullah Shaikh
College of Computer Science and Information Systems,
Najran University,
Najran, Saudi Arabia
and
The Maersk-McKinney Moller Institute,
University of Southern Denmark,
Odense, Denmark

Uffe Kock Wiil
The Maersk-McKinney Moller Institute,
University of Southern Denmark,
Odense, Denmark

***Abstract*—Model defects are a significant concern in the Model-Driven Development (MDD) paradigm, as model transformations and code generation may propagate errors present in the model 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.**

Current approaches to this problem have an exponential worst-case run time. In this paper, we propose a slicing technique which breaks a model into several independent submodels from which irrelevant information can be abstracted to improve the scalability of the verification process. We consider a specific static model (UML class diagrams annotated with unrestricted OCL constraints) and a specific property to verify (satisfiability, i.e., whether it is possible to create objects without violating any constraints). The definition of the slicing procedure ensures that the property under verification is preserved after partitioning. Furthermore, the paper provides an evaluation of experimental results from a real-world case study.

***Keywords*—MDD; UML; OCL; Model Slicing; Efficient Verification**

I. Introduction

Model-Driven Development (MDD) is a methodology widely used in the process of software development. The focus of MDD is on the use of models which can be transformed into code to save software developers time and effort. Transformation and code generation from models may spread errors in the code if the models are not verified, however.

There are formal verification tools for automatically checking correctness properties of models, but the lack of scalability of such tools is a serious problem. Addressing this problem is the goal of this paper.

At present, we face efficiency problems when verifying Object Constraint Language (OCL) constraints of complex Unified Modeling Language (UML) class diagrams. 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 speed-up in the verification process. We focus on analysis of static elements of a software specification, modelled 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* [9], [3], [37] and whether it is possible to instantiate the model without violating any integrity constraints. Constraints can be either textual OCL invariants or graphical restrictions like multiplicities of association ends.

This property is important because it can identify inconsistent models, but also 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 and vice versa. For instance, the *redundancy* of an integrity constraint C can be expressed as a satisfiability test: if we change the integrity constraint to $\neg C$ and the model is still satisfiable, this means that C is not redundant as it effectively avoids at least one undesired instance.

Furthermore, the addition of unrestricted[1]OCL constraints makes the problem undecidable. For example, reasoning on UML class diagrams is EXPTIME-complete [5] and, when general OCL constraints are allowed, it becomes undecidable.

In order to provide practical and workable solutions, tools for formal verification of UML/OCL class diagram must consider several aspects of the verification problem pragmatically: the desired degree of automation (fully automatic or user-guided?), the desired degree of completeness (conclusive answer for any input model?), and the degree of expressiveness allowed in OCL constraints.

Current solutions for checking satisfiability employ formalisms such as description logics [3], higher-order logics [8], database deduction systems [4], linear programming [35], SAT [21][2], or constraint satisfaction problems (CSP) [5], [9]. Each method provides different trade-offs in terms of decidability, completeness, expressiveness, and efficiency, which depend on the underlying formalism and tool support. All the approaches which support general OCL constraints share a common drawback, however: 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

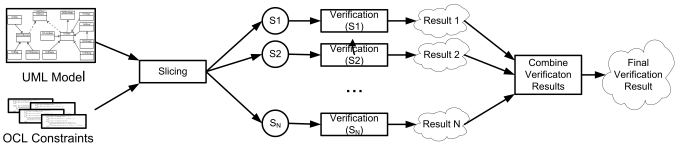


Fig. 1. High-level description of the slicing process.

of OCL constraints. This complexity is a serious limitation for the scalability of these techniques and their application in large-scale class diagrams.

A review of sample UML/OCL models highlights two observations which are relevant to this problem. First, models 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. Similarly, some integrity constraints regarding the multiplicity of association ends may be abstracted as well, e.g. an association end with a multiplicity of * does not constrain the model in any way which affects its satisfiability. 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 unnecessary model elements are abstracted. The structure of the class diagram (associations and class hierarchies) and the OCL invariants (abstract syntax tree) 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.

In contrast, there are a few verification and validation tools and techniques that verify the model properties and finds valid objects of the class diagrams [1], [20], [8], [17]. These tools and techniques are, however, inefficient (high Central Processing Unit and memory consumption) and unable to verify large UML/OCL class diagrams. The efficiency analysis of a few UML/OCL tools can be found in [41]. Therefore, the general question addressed in this paper is how we can improve the efficiency of the verification process for complex UML/OCL class diagrams.

A few hypotheses can be derived from the above discussion:

1) H1. Model slicing can be implemented in existing verification tools independent of their formalism.
2) H2. Model slicing will reduce the verification time.
3) H3. Model slicing enables verification of certain types of UML/OCL class diagrams that cannot be verified with current tools.

II. Contributions

This paper proposes a slicing technique for complex UML/OCL class diagrams which have a high worst-case computational complexity. This slicing technique is called the UML/OCL Slicing Technique (UOST). High worst-case computational complexity represents the amount of time in which the information in the model will be processed. The slicing procedure is based on breaking a complex UML/OCL model into several independent submodels where all irrelevant components of the model are abstracted from the complex hierarchy. The defined slicing procedure ensures that if all submodels are satisfiable then the entire model is satisfiable. If the model is unsatisfiable then some submodel is also unsatisfiable. The contributions of this paper are as follows:

1) A slicing procedure for a disjoint set of submodels
2) A procedure for analysis of OCL constraints
3) A procedure for detection of trivially satisfiable constraints
4) A procedure for analysis of UML class diagram
5) Application of the slicing technique in a real-world case study Digital Bibliography and Library Project (DBLP) conceptual schema
6) Experimental results in UMLtoCSP (UOST) [39] and in Alloy [20]

The major contribution of this paper, however, is improvement of the scalability of verification of UML class diagrams with OCL constraints. The presented slicing technique slices the model before it passes to any satisfiability analyser/engine, rather than passing a complete or complex model to any verification engine. We slice the model in the memory before checking satisfiability and this is the major reason for speed-ups in verification. In this paper, the experiments are conducted in Alloy [20] and UMLtoCSP [9]. The underlying satisfiability analyser used in Alloy is the Kodkod model finder and a variety of SAT solvers [47] whereas UMLtoCSP uses CSP [10]. With the help of the slicing technique these satisfiability analysers receive the original model as several independent submodels and check the satisfiability independently, which causes drastic speed-up in verification time. Verification time is totally dependent, however, on how fast these satisfiability analysers can find the valid objects as per conditions given in OCL constraints.

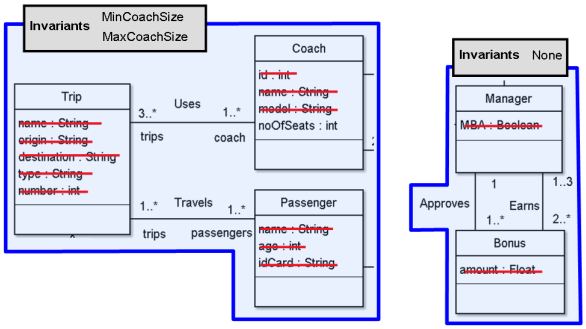
Previously, we have proposed a slicing technique for a non-disjoint set of submodels [41], [40], however, this paper extends our work published in ASE 2010 [38] with the contributions given in sections “Contributions”, VI, and VII. We have implemented slicing techniques for disjoint and non-disjoint sets of submodels in UMLtoCSP (UOST) [39]. Furthermore, comparison with other tools and techniques has been discussed in [41].

III. Overview of Proposed Slicing Technique

The input for the slicing procedure is a UML class diagram annotated with OCL invariants. Figure 2 presents a

[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.

[2] “SAT stand for ‘satisfiablity’: a solution to a boolean formula is an assignment of values to the formula’s boolean variables that ‘satisfies’ the forumula”[21].



(a) Slice 1 (b) Slice 2

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

class diagram that will be used as an example modelling 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 verification process will be either : 'satisfiable' or 'unsatisfiable'. In the 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 [9], [3], [37]. 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 to be 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 for the example system. 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 also whether *all* slices are strongly satisfiable. On the other hand, if we are checking weak satisfiability, it is sufficient to ensure that *at least one* slice is weakly satisfiable.

A. Preserving Satisfiability

The fundamental requirement of the slicing algorithm 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. Disjoint fragment contains different classes for each slice. 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 slice[s] will be weakly unsatisfiable.

A class diagram can be unsatisfiable for several reasons. First, it is possible that the model provides inconsistent conditions for 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, e.g., self.x = 7. Finally, it is possible that unsatisfiability arises from a combination of both factors, e.g., the values of some attributes require a certain number of objects to be created which contradict other restrictions.

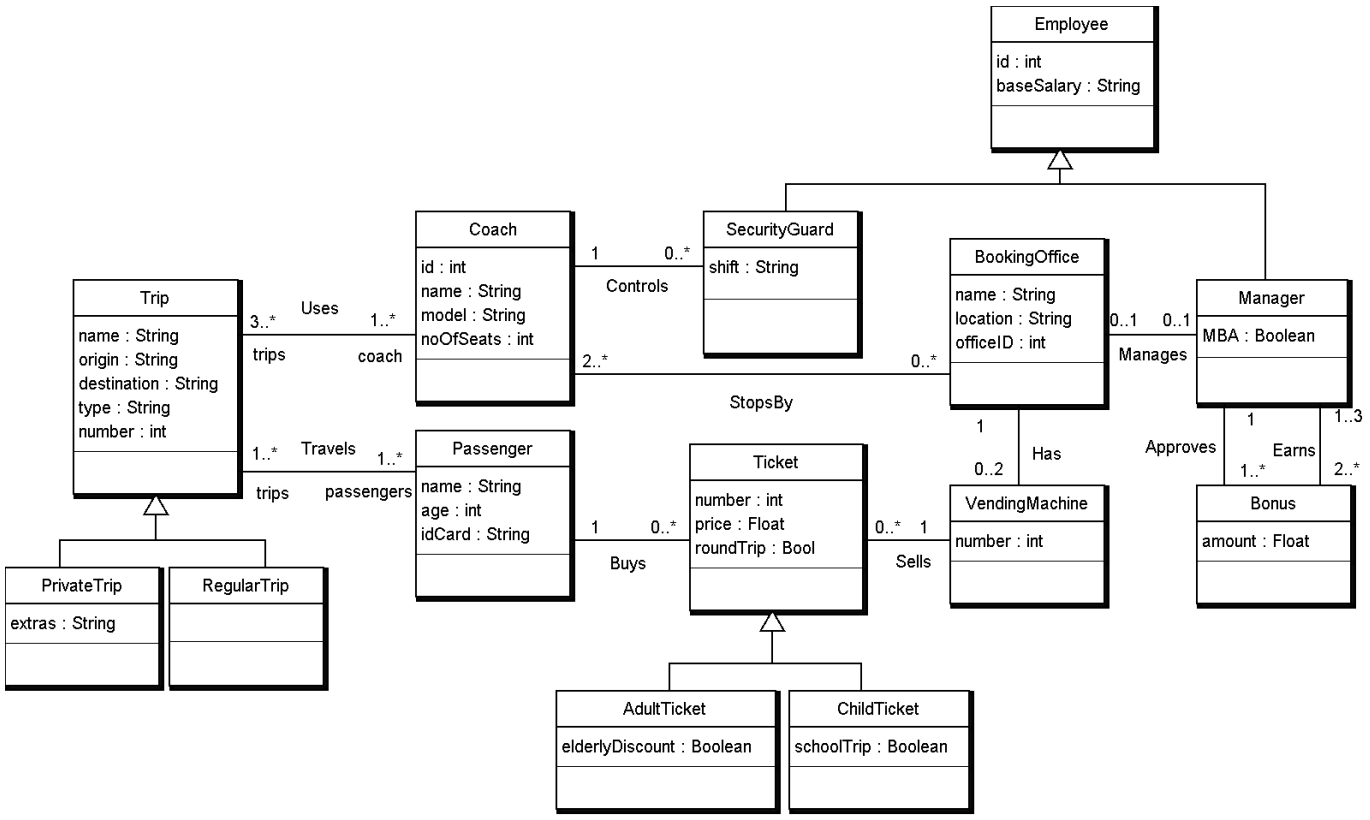
To sum up, an unsatisfiable model contains an unsatisfiable textual or graphical constraint or an unsatisfiable interaction between one or more textual or graphical constraints. To ensure that unsatisfiability is propagated in the slices, three conditions should be guaranteed:

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.

The procedure presented in this paper guarantees all conditions (1 to 3). Before slicing, the class diagram and integrity constraints are analysed to detect unconstrained elements, constraints which do not affect satisfiability and constraints which cannot interact adversely with other constraints. In order to provide this assurance, it is necessary to analyse the UML class diagram before slicing in order to know what can be partitioned and abstracted and what should be kept together. The analysis is performed at two levels, a syntactic analysis of the OCL constraints and a structural analysis of the UML class diagram:

- A traversal of the syntax tree of each OCL constraint identifies which classes, attributes, and navigations are used. Additional analysis identifies trivial constraints and constraints that can be checked independently.
- The analysis of the UML class diagram reveals dependencies among the number of objects in each class, like inheritance hierarchies or multiplicity constraints of association/aggregation ends.

The following sections describe the analysis of OCL invariants and the UML class diagram. The combination of the verification results from each slice is straightforward (either all slices have to be satisfiable or at least one has to be for strong



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

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

and weak satisfiability, respectively) and will not be detailed further.

IV. Analysis of OCL Constraints

OCL allows the definition of *expressions* on UML class diagrams. 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.

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() and 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 class diagrams.

This section describes how to analyse 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.

A. Constraint Support

The *support* of an OCL expression is the subset of classes of the class diagram referenced by the expression. For invari-

TABLE I. SUPPORT, ATTRIBUTES, AND NAVIGATIONS IN THE RUNNING EXAMPLE.

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

ants, the support describes the set of classes restricted by the constraint. This information will be used to identify classes that appear together in the same constraint and therefore must be analysed within the same slice. Formally, the support of an expression E and the supertypes of 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 steps of the analysis 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 I summarises all these data for the invariants of the running example.

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 can be used to compute the clusters:

- Compute the support of each invariant.
- Initially, each constraint is located in a different cluster.
- Select two constraints x and y with non-disjoint supports (i.e., support(x) $\cap$ support(y) $\neq \emptyset$) and located in different clusters, and merge those clusters.
- Repeat the previous step until all pairs of constraints with non-disjoint support belong to the same cluster.

Using this procedure and the information from Table I, we can identify three clusters in our model: invariants MinCoachSize, MaxCoachSize, CorrectTripDestination and NonNegativeAge (support: Coach, Trip, Passenger); invariant MachineNumber (support: VendingMachine, BookingOffice) and invariant UniqueTicketNumber (support: Ticket). In the following sections, however, we describe additional analysis that can abstract constraints before this clustering, simplifying the problem that has to be verified.

B. 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 overall solution. Intuitively, if there is a constraint on an attribute which is not used anywhere else in the model, we can split the verification problem into two separate subproblems: checking that the constraint on the attribute is feasible and verifying the rest of the system. This section will present the techniques which identify such local constraints.

TABLE II. EXAMPLES OF LOCAL AND GLOBAL INVARIANTS.

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()

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: (1) it does not use navigations through associations, (2) it does not call allInstances(), (3) it does not use attributes defined in a superclass, (4) it does not call any global query operation, and (5) it does not perform any type check or type conversion operation. Table II shows some examples of local and global expressions written in the context of class *Trip*.

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: (1) the attributes used in global expressions plus (2) 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 (1) attributes outside those mentioned in the expression or (2) the 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 analysed separately from the rest of the model. The division into subproblems is as follows:

- 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.

C. 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 considering 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's value must be unique, e.g. Type::allInstances() $->$isUnique(obj | obj.attr). If the attribute is of Integer, Float, or String type 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. Therefore, the attribute and the constraint can be safely removed from the problem without affecting its satisfiability.

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 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 the right value of *attrib*. The conditions for *expression* (no self-references, no undefined values) guarantee that the evaluation always computes a feasible value for *attrib*. In some cases, derived value constraints involving recursive query operations may render/make a model to become unsatisfiable. In that case, if there is any repetition in the same constraint (i.e., using recursive query operations) with the same pattern *self.attrib op expression* will not be counted as a derived value constraint. The recurrence of values of same objects make a model unsatisfiable. Therefore, if there is any repetition in OCL constraint will not be removed from the problem. Table III briefly summarises the patterns and conditions where the column *Pattern* shows the possible expressions and the column *Condition* illustrates the criteria for including the corresponding pattern.

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

TABLE III. PATTERNS WITH CONDITIONS.

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 both A and B are satisfiable and have no interdependencies.
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.

V. Analysis of UML Class Diagrams

In this section, we will consider a UML class diagram composed of binary associations and inheritance relations. The features of class diagrams like associative classes or n-ary associations can be expressed in terms of binary associations (and potentially additional OCL constraints) [18].

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 analysed 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) but also conservative (related vertices will not be separated under any circumstances).

In order to define these relationships, we will use an auxiliary graph-based representation called a *flow graph*. A flow graph is a labelled directed pseudograph, i.e., there can be arcs from a 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 $X \xrightarrow{n} Y$ has means 'if there is an object in class X, at least n objects of class Y must exist'. Using this definition, there is an arrow $X \xrightarrow{n} Y$ 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 at Y is n.

Arcs with a label of zero can be removed because they are not imposing any constraint. Multiple arcs between two vertices can be replaced by a single arc labelled with maximum label. 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

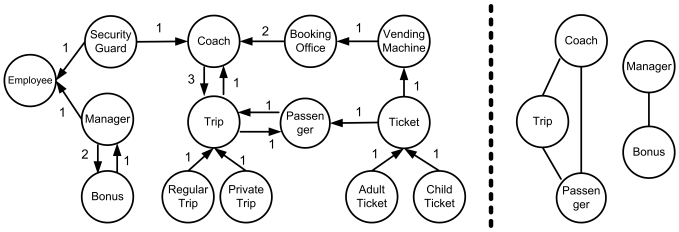


Fig. 4. Flow graph (left) and dependency graph (right) for the running example.

a cyclic dependency and it is therefore a possible source of unsatisfiability. Any cycle where the maximum label is one is inherently satisfiable, and it will be called *safe*, but cycles where (1) the maximum label ≥ 2 and (2) there are two or more participating associations/inheritance 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) whereas 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 be 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*.

It is possible to extract its connected components from the dependency graph. Each component defines a slice of the class diagram that can be analysed independently: the set of classes from the class diagram, the set of associations and the 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 that 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. As regards weak satisfiability, however, 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 that (1) X does not appear in the dependency graph and (2) 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 on X can be populated even if no class of the dependency graph can be populated. In our running example, class *Employee* is the only class which exhibits this trait.

VI. DBLP Conceptual Schema

This section demonstrates the application of the slicing algorithm in a real-world case study: the conceptual schema of the DBLP system, modelled as a UML class diagram. It is a computer science bibliographical website, dating from the 1980s [15]. 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. This case study is interesting for our problem since it has complex invariants and is a real-world case study. Therefore, we applied our slicing approach to this DBLP case study in order to show that our method works for external case studies and can improve the efficiency of the verification process.

Figure 5 introduces the DBLP class diagram that will be used as an example to demonstrate slicing. Several integrity constraints are defined as OCL invariants which we classify in three types of categories: *key constraints, derived value constraints,* and *indispensable integrity constraints*. Table IV describes the list of constraint names, constraint supports, and the category of constraints (key, derived value or indispensable). The key constraints and derived value constraints are considered as trivially satisfiable patterns and can safely be removed from the problem in order to improve the efficiency of the verification process without considering its satisfiability. Another category of integrity constraint is the indispensable integrity constraints which are neither key constraints nor derived value constraints and therefore cannot be abstracted. These types of OCL invariants cannot be removed because their attributes are constrained by more than one invariant which affects their satisfiability. For example, *self.journalVolume −>isUnique(volume)* could be considered

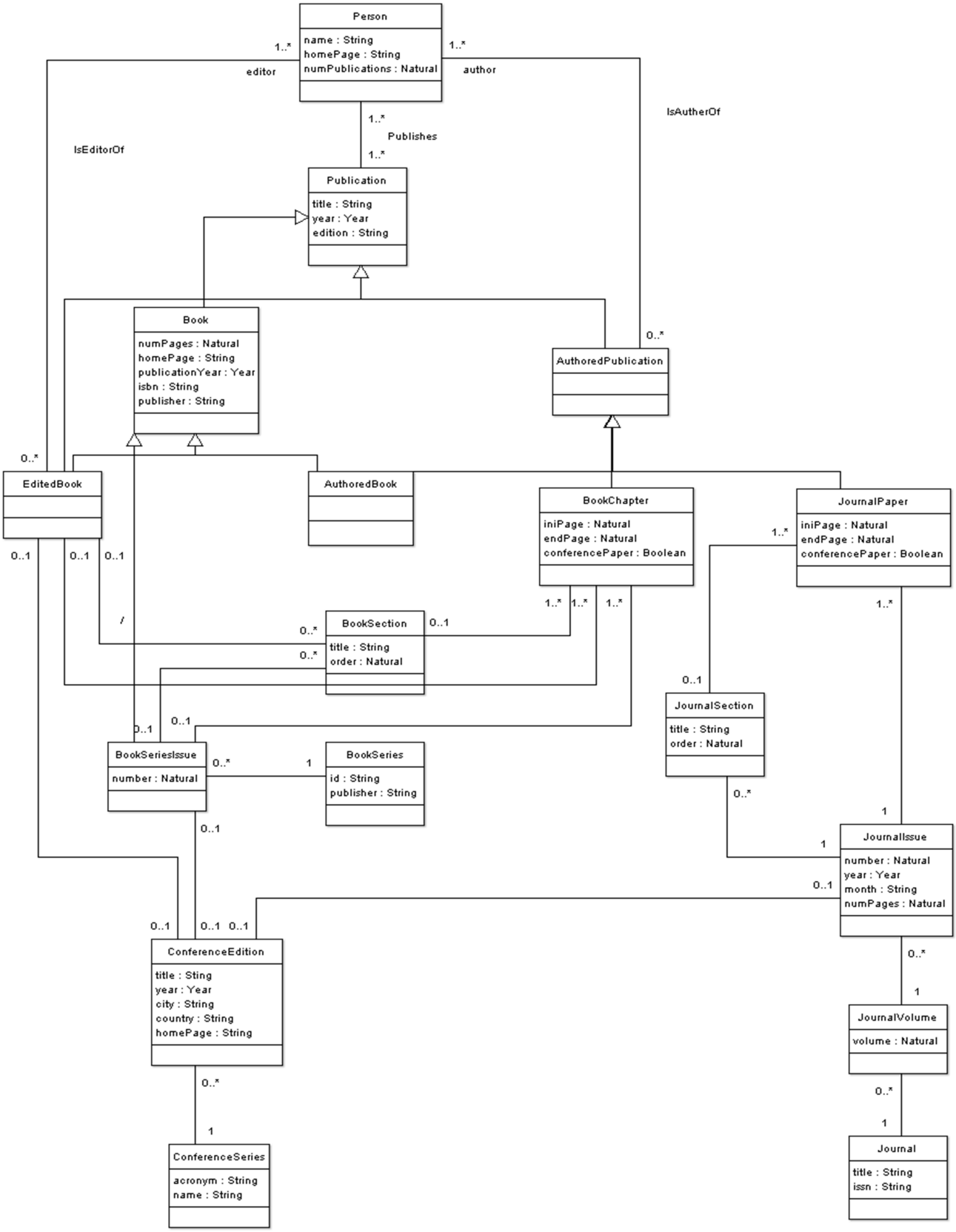


Fig. 5. DBLP class diagram.

a key constraint, but it is not a key constraint because there is another constraint which affects the same attribute such as *self.journalVolume −>sortedBy(volume).volume = sequence* { *1..self.journalVolume −>size()* }.

A. Slicing DBLP

The input model is a DBLP case study annotated with 26 OCL invariants. After the elimination of key constraints and derived value constraints, we have 10 other integrity constraints whose satisfiability needs to be checked. Out of these 10 constraints, there are two *local* and eight *global* constraints. In order to identify the slices, we need, first of all, to compute a flow graph of the DBLP case study that captures the dependencies of the model elements, then the connected components of the graph will be computed respectively. In a flow graph, each vertex is a class and the arcs are non-negative integers showing the association between one vertex and another. Arcs with a label 0 are removed from the DBLP class diagram because they are not restricting any OCL invariant except ConferenceEdition-EditedBook, ConferenceEdition-BookSeriesIssue, and ConferenceEdition-JournalIssue. Figure 6 illustrates the flow graph of the DBLP class diagram after the elimination of the unnecessary arcs. The next step involves the detection of cycles among the vertices. Because of cyclic dependency, it is possible that the model may become unsatisfiable. In the case of DBLP case study, all cycles have the maximum label 1 and therefore all will be deemed safe. There are two cycles: Person-Publication and JournalPaper-JournalIssue which exists in a DBLP conceptual schema; however, no unsafe cycle exists. Applying all this information, we create the dependency graph. Initially, the classes constrained by the OCL invariants will be identified. Second, the vertices corresponding to these classes will be added. For example, there is an arc between vertex JournalPaper and JournalIssue in the graph; however, the arc between JournalVolume and JournalIssue is not included in the graph. Using the path from JournalIssue to JournalVolume, an arc between JournalVolume and Journal can now be added to the flow graph. Edges with a multiplicity 0 which are not navigated through any constraint are not depicted in the diagram.

Finally, the connected components will be extracted from the flow graph. Each of these single components is a slice and is composed of a set of classes, associations, and invariants. Figure 7 describes two resulting slices, whose satisfiability must be checked. These slices are made on the results from indispensable integrity constraints which are not trivially satisfiable. Table V summarises the constraint name, support, required classes, and submodel for the indispensable integrity constraints. As the DBLP case study involves complexity, there is only the possibility of slicing the conceptual schema between JournalIssue and JournalVolume. Consequently, five classes are eliminated from the hierarchy, i.e., Person, BookSection, BookSeries, JournalSection, and ConferenceSeries.

The detection of trivially satisfiable invariants means that some attributes like title, name, or city which were constrained before by an invariant can now be simply abstracted as can be seen in Table VI. The description of the columns is as follows: column *name* describes the names of the classes; column *attribute* outlines the list of attributes for the entire DBLP class diagram; column *restricted attributes* describes those attributes

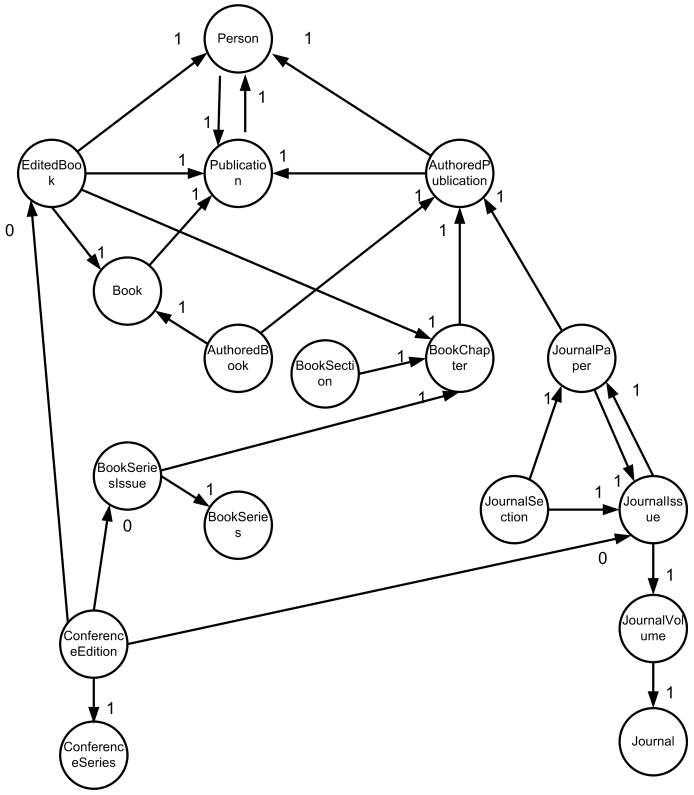


Fig. 6. DBLP flow graph.

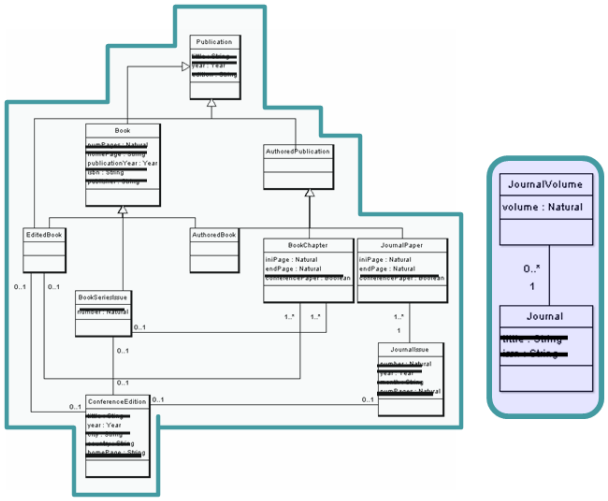


Fig. 7. Submodels 1 and 2 of DBLP.

which are constrained by indispensable integrity constraints; column *unrestricted attributes* refers to those attributes which are not constrained at all and, finally, column *unrestricted after removal* shows those constraints which were constrained by key and derived value constraints. In the end, the only attributes which need to be considered are those from column *restricted attributes*.

VII. Experimental Results

In this section, we measure the speed-up achieved by implementing the slicing technique in two different tools.

TABLE IV. NAME, SUPPORT, AND CATEGORY IN THE DBLP CLASS DIAGRAM.

Constraint Name	Constraint Support	Is Key or Is Derived Value or Indispensable Constraints
nameIsKey	Person (name)	Is Key
isbnIsKey	Book (isbn)	Is Key
idIsKey	BookSeries (id)	Is Key
BookSeriesAndNumber IdentifyBookSeriesIssue	BookSeries (number)	Is Key
issnIsKey	Journal (issn)	Is Key
titleIsKey	Journal (title)	Is Key
editedBookWithout Repetitions	EditedBook, BookSection (title)	Is Key
bookSeriesIssueWithout Repetitions	BookSeriesIssue, BookSection (title)	Is Key
journalSectionWithout Repetitions	JournalSection, JournalPaper (title)	Is Key
bookSectionWithout Repetitions	BookSection, BookChapter (title)	Is Key
journalVolumeAndNumber IdentifyJournalIssue	JournalVolume, JournalIssue (number)	Is Key
journalIssueAndTitle IdentifyJournalSection	JournalIssue, JournalSection (title)	Is Key
nameIsKey	ConferenceSeries (name)	Is Key
titleIsKey	ConferenceEdition (title)	Is Key
journalAndVolume IdentifyJournalVolume	Journal, JournalVolume (volume)	Indispensable
correctPagination	BookChapter (iniPage, endPage)	Indispensable
correctPagination	JournalPaper (iniPage, endPage)	Indispensable
correctPagination	JournalIssue, JournalPaper (iniPage, endPage)	Indispensable
correctPagination	EditedBook, BookChapter (iniPage, endPage)	Indispensable
correctPagination	BookSeriesIssue, BookChapter (iniPage, endPage)	Indispensable
consecutiveVolumes	Journal, JournalVolume (volume)	Indispensable
compatibleYear	EditedBook, ConferenceEdition (year), Book (publicationYear)	Indispensable
compatibleYear	BookSeriesIssue, ConferenceEdition (year), Book (publicationYear)	Indispensable
conferenceIsPublished	ConferenceEdition, EditedBook, BookSeriesIssue, JournalIssue	Indispensable
theSamePublisher	Book, BookSeriesIssue, BookSeries (publisher)	Is Derived Value
compatibleYear	JournalIssue (year), ConferenceEdition (year)	Is Derived Value

TABLE V. CONSTRAINT NAME, SUPPORT, TIGHTLY COUPLED CLASSES AND SUBMODEL FOR INTEGRITY CONSTRAINT.

Constraint Name	Constraint Support	Tightly Couped Classes	Submodel
correctPagination	BookChapter (iniPage, endPage)	BookChapter, AuthoredPublication Publication	1
correctPagination	JournalPaper (iniPage, endPage)	JournalPaper,JournalIssue AuthoredPublication, Publication, JournalVolume,Journal	1
correctPagination	JournalIssue, JournalPaper (iniPage, endPage)	JournalPaper,JournalIssue AuthoredPublication, Publication, JournalVolume,Journal	1 1
correctPagination	EditedBook, BookChapter (iniPage, endPage)	EditedBook,BookChapter, AuthoredPublication	1
correctPagination	BookSeriesIssue, BookChapter (iniPage, endPage)	BookSeriesIssue,BookSeries, AuthoredPublication Publication, Book,BookChapter	1
compatibleYear	BookSeriesIssue, Book (publicationYear), ConferenceEdition (year)	BookSeriesIssue,Book BookChapter, AuthoredPublication, Publication, ConferenceEdition	1
compatibleYear	EditedBook, Book (publicationYear), ConferenceEdition (year)	EditedBook,Book Publication, ConferenceEdition BookChapter, AuthoredPublication	1
conferenceIsPublished	ConferenceEdition, EditedBook, BookSeriesIssue, JournalIssue	ConferenceEdition, EditedBook, Book BookSeriesIssue, journalIssue, Publication, JournalPaper, AuthoredPublication	1
journalAndVolume IdentifyJournalVolume	Journal, JournalVolume (volume)	Journal,JournalVolume	2
consecutiveVolumes	Journal, JournalVolume (volume)	Journal,JournalVolume	2

Initially, we had developed a prototype implementation of the slicing procedure on top of the tool UMLtoCSP [9]. UMLtoCSP transforms verification problems involving 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 to be verified. Figure 8 presents the method of applying slicing technique in UMLtoCSP. Further discussion on translation of UML/OCL class diagrams, transformation of classes and, definition of correctness of properties can be found in [10]. Second, we slice the conceptual schema of the DBLP programmed in Alloy [20] in order to show the drastic speed-up. Alloy is a widely-used structural modelling language based on first-order logic (FOL) and can generate instances of invariants and simulate the execution of operations. The purpose behind showing the results in the Alloy specification is to demonstrate that the developed slicing technique is neither tool dependent nor formalism dependent. It can be implemented in any verification-driven UML/OCL tool. These two cases support hypothesis 1 (H1).

A. Slicing in UMLtoCSP

In the first case, we compare the verification time of several UML/OCL class diagrams using (1) the original tool

TABLE VI. RESTRICTED AND UNRESTRICTED ATTRIBUTES FOR DBLP CASE STUDY.

Class Name	Attributes	Restricted Attributes	Unrestricted Attributes	Unrestricted Attributes After Removal
Person	name: String homePage: String numPublications: Natural	None	homePage: String numPublications:Natural	name: String
Publication	title: String year: Year edition: String	None	title:String year:Year edition:String	None
Book	numPages: Natural homePage: String publicationYear: Year isbn: String publisher: String	publicationYear:Year	numPages:Natural homePage:String	isbn:String publisher:String
EditedBook	None	None	None	None
AuthoredBook	None	None	None	None
AuthoredPublication	None	None	None	None
BookChapter	iniPage: Natural endPage: Natural conferencePaper:Boolean	iniPage: Natural endPage: Natural	conferencePaper:Boolean	None
BookSection	title: String order: Natural	None	order:Natural	title:String
BookSeriesIssue	number: Natural	None	None	number:Natural
BookSeries	id: String publisher: String	None	None	id:String publisher:String
ConferenceEdition	title: String year: Year city: String country: String homePage: String	year: Year	city: String country: String homePage: String	title: Sting
ConferenceSeries	acronym: String name: String	None	acronym:String	name: String
JournalPaper	iniPage: Natural endPage: Natural conferencePaper:Boolean	iniPage: Natural endPage: Natural	conferencePaper:Boolean	None
JournalSection	title: String order:Natural	None	order:Natural	title: String
JournalIssue	number: Natural year: Year month: String numPages: Natural	None	number: Natural month: String numPages: Natural	year: Year
JournalVolume	volume:Natural	volume: Natural	None	None
Journal	title: String isbn:String	None	None	title: String isbn:String

TABLE VII. DESCRIPTION OF THE UML/OCL BENCHMARKS.

Example	Classes	Associations	Attributes	Invariants	Strongly Satisfiable?
Atom-Molecule	2	1	6	1	Yes
Paper-Researcher	2	2	5	4	No
Coach	13	13	27	6	Yes
Production System	50	30	72	5	Yes
Company	100	100	100	100	Yes
DBLP Conceptual Schema	17	19	38	26	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

UMLtoCSP and (2) the tool UMLtoCSP (UOST) with slicing [39]. UMLtoCSP (UOST) is developed in JAVA and "the basic approach behind the UMLtoCSP (UOST) tool is a model slicing technique that enables efficient verification of UML/OCL class diagrams. The tool can verify different sets of properties for UML/OCL class diagrams with disjoint and non-disjoint sets of slicing. The features strong satisfiability and weak satisfiability are same as in UMLtoCSP [9]. However, other new features in UMLtoCSP (UOST) are:

Strong satisfiability: the class diagram should have a legal instance for at least one object from each class and a link from each association.

Weak satisfiability: the class diagram should have a legal instance/object which is non-empty, i.e., it contains at least one object from some class.

Remove attributes: for weak or strong satisfiability, unrestricted attributes can be removed from the class diagram.

Non-disjoint slicing: slicing of a class diagram with non-disjoint sets of submodels.

Disjoint slicing: slicing of a class diagram with disjoint sets of submodels.

Show specific invariants: detection of failing submodel(s) in disjoint slicing and a specific unsatisfiable invariant(s) in non disjoint slicing"[39].

In each example verified in UMLtoCSP (UOST), the property to be verified has strong satisfiability. Table VII describes

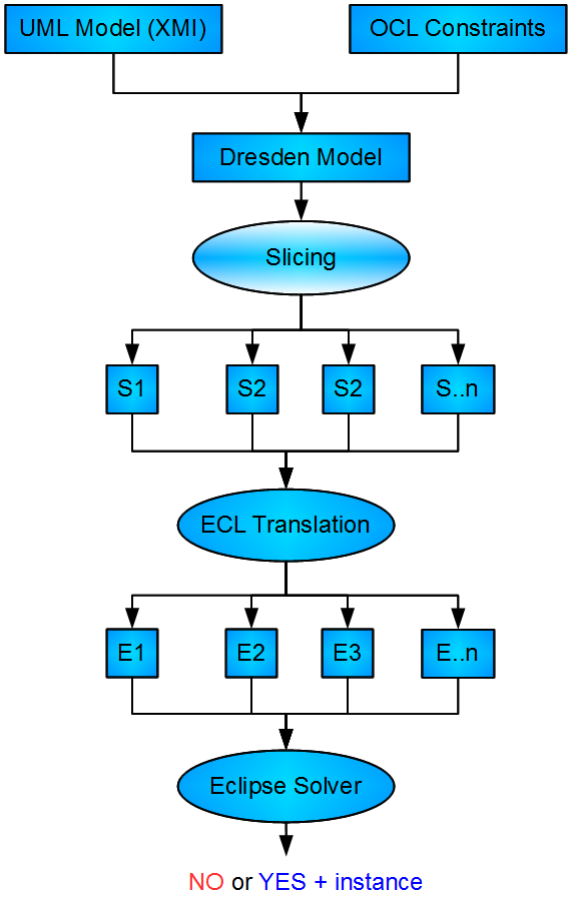
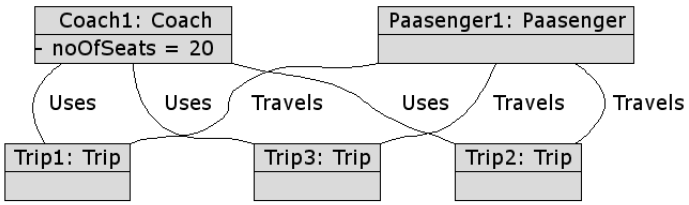
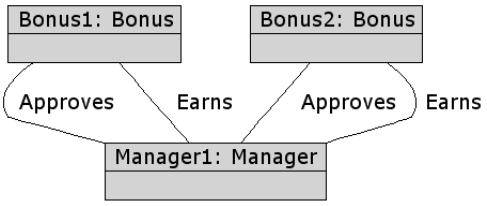


Fig. 8. Slicing procedure in UMLtoCSP.



(a) Submodel 1 of 'Model Coach'



(b) Submodel 2 of 'Model Coach'

Fig. 9. UMLtoCSP output of model coach.

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, parameters of large value will make the comparison more favourable in terms of slicing, as abstracting attributes and classes will cause a larger reduction of the solution space. In our analysis, we considered small but reasonable values for parameters: at most four 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 VIII 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 was set at two hours (7200 seconds). For each model, 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). Figure 9 shows the UMLtoCSP output, i.e., object diagram of 'Model Coach'.

The first conclusion is that slicing is a very fast procedure even in diagrams with hundreds of classes and it is a formalism independent technique. As expected, the effectiveness of the technique depends on the specific model analysed: small models and models where UMLtoCSP has already been performed will 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. In models where slicing manages to partition the model and abstract attributes, however, the speed-up reaches several orders of magnitude. Therefore, its success will depend on the type of model where it is applied. Small models which have been manually preprocessed for verification will gain little from slicing. Models created for other purposes or models generated through automatic transformation can, however, benefit greatly from the application of slicing. Therefore, hypothesis 2 (H2) is supported.

The tiny overhead introduced by slicing and the tool-independent nature of this approach are additional reasons in favour of adding slicing to existing formal verification toolkits. A larger real-world case study where further benefits of slicing are illustrated in different formalism (i.e., SAT) is presented in Section "Slicing Alloy Specification (DBLP)".

B. Slicing in Alloy

In the second case, we have applied the slicing technique on a few examples programmed in the Alloy specification in order

TABLE VIII. DESCRIPTION OF EXPERIMENTAL RESULTS (CASE 1).

Example	OVT	Slices	Attr	ST	SVT	Times
Atom-Molecule	0.03s	1	3	0.00s	0.03s	0x
Paper-Researcher	0.04s	1	0	0.00s	0.04s	0x
Coach	5008.76s	2	26	0.00s	0.15s	33392x
Production System	3605.35s	4	59	0.02s	0.03s	72107x
Company	0.08s	1	0	0.00s	0.08s	0x
DBLP Conceptual Schema	Time-out	2	18	0.19s	0.37s	Not verifiable without slicing
Script 1	Time-out	2	117	0.02s	0.03s	Not verifiable without slicing
Script 2	Time-out	4	509	0.09s	0.02s	Not verifiable without slicing
Script 3	Time-out	4	1009	0.29s	0.34s	Not verifiable without slicing
Cycle 1	Time-out	1	10	0.00s	Time-out	Not Available
Cycle 2	Time-out	1	100	0.00s	Time-out	Not Available

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

to prove that our developed slicing technique is neither tool-dependent nor formalism-dependent. To translate the models into the Alloy language, we have used the model finder to generate possible model instances, which proves satisfiability of the model.

We compare the verification time of UML/OCL class diagrams using the Alloy analyser with and without the slicing technique. Table IX describes the set of benchmarks used for our comparison: the number of classes, associations, invariants, and attributes.

Tables X, XI, and XII summarise the experimental results obtained with the Alloy analyser before and after slicing, running on an Intel Core 2 Duo Processor 2.1Ghz with 2Gb of RAM. Each table represents the results as described in the benchmark (Table IX). The execution time is largely dependent on the defined scope. Therefore, in order to analyse the efficiency of verification, the scope is limited to four. The Alloy analyser will examine all the examples with up to four objects, and try to find one that violates the property. For example, specifying scope four means that the Alloy analyser will check models whose top level signatures have up to four instances.

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, 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 speed-up shows the efficiency obtained after the implementation of the slicing approach.

Previously, with no slicing, it took 200 ms (scope 4) for the execution of 'University' and 254 ms (scope 4) for 'ATM Machine'. With the UOST approach, it takes only 72 ms (scope 4) for 'University' and 24 ms (scope 4) for 'ATM Machine'. It is an improvement of 64% and 90%, respectively. In addition, the improvement can also be achieved for larger scopes as well.

C. Slicing Alloy Specification (DBLP)

We have also applied our slicing technique to the DBLP structural schema programmed in the Alloy specification. The schema that we slice with our approach defines 26 integrity constraints. The approach is manually implemented in the

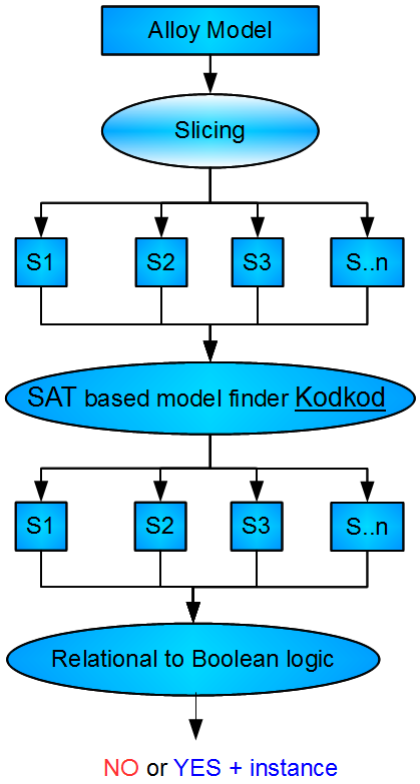


Fig. 10. Slicing procedure in Alloy analyser.

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 used for slicing in Alloy to check the advantages of slicing. The execution time is largely dependent on the defined finite scope and therefore, in order to analyse the efficiency of verification, we limit the scope to a minimum of two and a maximum of 22. Figure 10 shows the general procedure of implementation of the slicing technique in the Alloy analyser.

After application of the technique, two submodels are obtained: submodel 1 consists of 10 classes annotated with eight OCL constraints and submodel 2 comprises two classes annotated with two OCL constraints. Table XIII summarises the experimental results obtained with the Alloy analyser before and after slicing, running on an Intel Core 2 Duo

TABLE IX. DESCRIPTION OF THE EXAMPLES.

Example	Classes	Associations	Attributes	Invariants
Atom-Molecule	3	3	4	4
University	5	4	9	7
ATM Machine	50	51	51	7

TABLE X. SLICING RESULTS IN ALLOY FOR ATOM-MOLECULE EXAMPLE.

Before Slicing				After Slicing			
Scope	TT	ST	TT+ST	STT	SST	STT+SST	Speedup %
2	115ms	70ms	185ms	109ms	56ms	165ms	10%
3	138ms	76ms	214ms	117ms	65ms	182ms	15%
4	153ms	100ms	253ms	119ms	70ms	189ms	25%

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

TABLE XI. SLICING RESULTS IN ALLOY FOR UNIVERSITY EXAMPLE.

Before Slicing				After Slicing			
Scope	TT	ST	TT+ST	STT	SST	STT+SST	Speedup %
2	67ms	50ms	117ms	24ms	10ms	34ms	29%
3	92ms	56ms	148ms	35ms	30ms	65ms	56%
4	134ms	66ms	200ms	39ms	33ms	72ms	64%

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

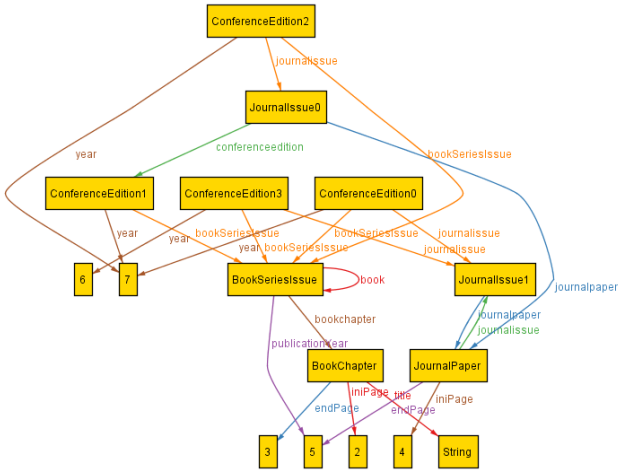
TABLE XII. SLICING RESULTS IN ALLOY FOR ATM MACHINE.

Before Slicing				After Slicing			
Scope	TT	ST	TT+ST	STT	SST	STT+SST	Speedup %
2	20ms	46ms	66ms	5ms	8ms	13ms	81%
3	83ms	91ms	174ms	9ms	11ms	20ms	89%
4	96ms	185ms	254ms	13ms	11ms	24ms	90%

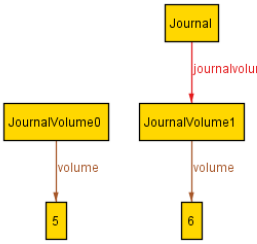
TT Translation Time **ST** Solving Time
STT Sliced Translation Time **SST** Sliced Solving Time

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 measured: i.e., 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.

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% and we have also achieved minimum 18% (scope 2) and maximum 80% (scope 22) speed-up which is marked progress in terms of total execution time. In addition, the improvement can also be achieved for larger scopes as well. For instance, we have conducted experiments with a maximum scope of 22, and therefore, at certain scopes, it is possible to reach up to 99% improvement. Without slicing, however, we could only run the analysis only up to a limited scope. Figure 11 shows the object diagram of DBLP in the form of submodel 1 and submodel 2 as output in the Alloy analyser. This experiment supports hypothesis 3 (H3) - that model slicing enables verification of certain types of UML/OCL class diagrams that cannot be verified with current tools.



(a) Submodel 1



(b) Submodel 2

Fig. 11. Alloy output of DBLP conceptual schema.

TABLE XIII. DESCRIPTION OF EXPERIMENTAL RESULTS (CASE 2).

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%
8	1132ms	567ms	1699ms	602ms	240ms	842ms	50%
9	1694ms	906ms	2600ms	787ms	302ms	1089ms	58%
10	2049ms	1149ms	3198ms	854ms	208ms	1062ms	66%
11	2751ms	1297ms	4048ms	1054ms	179ms	1233ms	70%
12	3934ms	1935ms	5869ms	1283ms	351ms	1634ms	72%
13	6361ms	2838ms	9199ms	1902ms	435ms	2337ms	75%
..	....	....	....	....	....	....	...
..	....	....	....	....	....	....	...
..	....	....	....	....	....	....	...
22	26034ms	11049	37083ms	6736ms	584ms	7320ms	80%

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

VIII. RELATED WORK

Slicing techniques can be classified according to two criteria: the *entity* to be sliced (e.g., a program, a UML model, an ontology, etc.) and the *goal* of the slicing process (e.g., synthesis, analysis, optimisation, visualisation, comprehension, etc.). 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 appended 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 [56], [46] 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, optimisation, verification and comprehension. Slicing has also been used in the analysis of the *architectural specifications* of a software system [43], [26], [57]. 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* [52], [51]. 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 used. The declarative modelling language Alloy plays a vital role in model verification and the complexity of declarative models is equal to that of the UML model with OCL constraints; therefore, the slicing in Alloy is needed. A novel optimisation technique based on program slicing for declarative models in order to perform efficient analysis is proposed. The algorithm works by partitioning slices for Alloy models into a case and derived slices. Afterwards, a satisfying instance of a base slice is generated to find the solution for the entire model. If the base slice is unsatisfiable then the entire model is unsatisfiable [53].

A challenging area is the testing of software product lines using SAT-based analysis. An interesting technique is proposed which incrementally generates tests for product lines. In this approach, the features of the program are the basis of incremental test cases. Afterwards, these features are converted into incremental test suites via transformation [50]. For model-based abstractions, a novel approach towards general automation for Model Driven Architecture (MDA) is introduced which is known as the FORMULA framework. FORMULA is a specification language and analysis tool that can be used to construct general MDA abstractions [22].

A further interesting angle of related work is systematic constraint-based test generation for programs and slicing for Alloy models. An interesting method for constraint-based test generation for programs is proposed which is based on organised generation of structurally complex test data from declarative constraints. The approach takes structurally complex data as an input, generates high-quality test cases and finds bugs in non-trivial programs [25]. Moreover, a novel approach which incrementally generates tests for product lines is proposed. In this work, test generation derives from the functions of the program [54]. A software system grows in size and complexity and therefore testing becomes a challenging task. In order to address the complexity issues, a novel approach to synthesising declarative specifications is presented. Partitioning is applied to enable efficient incremental analysis which defines a suite of optimisation in order to improve the analysis [49].

The most recent work on program slicing focuses on the reduction of the source code by program slicing before test generation [11]. This method is implemented in a tool called SNATE (Static Analysis and Testing). Furthermore, dynamic backward slicing for programs and the model transformation explores the idea of tracing the model transformations [48]. This work is based on program slicing for model transformations where the primary aim is to assess data and control dependencies.

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 [14], [36], [45] focus on pruning large ontologies to produce

smaller ontologies which are more manageable.

Slicing methods have also been proposed in the management of different types of UML models. *Context-free slicing* [24] 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 the goal. There is no discussion on the definition of suitable slicing criteria for verification. A different approach focusing on class diagram comprehension uses coupling metrics [27] to slice large models for visualisation. This type of approach would not be suitable 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 [29], [30]. A common representation, Model Dependency Graphs, is used to encode both types of diagrams. Again, the slicing criterion must be provided as an input to the algorithm.

Other similar work to our approach is the slicing of statecharts. *Slicing hierarchical automata for model checking* [23] presents an approach for slicing statecharts for the verification of properties. This research highlights the concept of slicing criteria for states and transitions. The algorithm is based on the removal of irrelevant hierarchies and concurrent states whereas our goal is the verification of specific properties using OCL constraints. We remove the OCL invariants to reduce the complexity of the model and then slice. The slicing criteria described in this paper are based on the scope of OCL constraints. Another piece of work related to our own is *System Verification through logic* (SV_tL) [55]. SV_tL provides a verification environment based on slicing for UML statecharts. It also removes irrelevant hierarchies in order to reduce the complexity of the verification for statecharts. The slicing criteria are based on dependency relations among states and transitions of the system.

Slicing of UML state machines is another type of related work [31]. This research provides a framework for creating smaller models for UML state machines given the fact that the behavior of the model should be the same. The method of slicing simplifies the model with the help of features. It is based on path predicates. Furthermore, *slicing of state-based models* [28], [12], [19], [2] classifies the segments of the model based on the element of interest. This approach presents a slicing technique that reduces the complexity of state-based models. The main use of slicing is for extended finite state machine (EFSM) models; however, the technique can also be applied to Specification and Description Language (SDL) models and statecharts.

Recent work on the slicing of UML models using model transformation has been presented by Kevin Lano [32], [33]. The purpose of slicing is to break the model into several sub-models for better analysis and understanding. The slicing technique is applied to UML class diagrams and state machines. The main goal of slicing is model transformation. Similar practical approach for model slicing is also proposed which is based on extraction of sub-models from original model to ease software visualization. The proposed methodology uses the idea of model based slicing of sequence diagrams to extract desired sub-models [42]. Further recent work is introduced by Wuliang Sun et al. [44] where authors invented model slicing for invariant checking, and applied a slicing technique to reduce the size inputs to improve efficiency of verification process in current tools. It is further proven that model slicing can drastically reduce the verification time. A general model-based slicing framework is also proposed that can be used to define both program and model slicing. The purpose of the framework is to construct slices written in a UML-like language [13].

The Kompren language to generate model slicers for Domain Specific Modeling Languages (DSMLs) is recommended for different purposes: for example, examining and model understanding. This work presents the model properties of the slices which can be extracted from different forms of the slicer [6], [7].

In contrast to these previous works, our paper describes a slicing criterion oriented towards the verification of satisfiability of UML/OCL class diagrams. We slice UML/OCL class diagrams before transformation. 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 optimisations have been considered extensively, for instance [16], [34], [52]. We suggest that slicing *before* the translation into a formalism like SAT or CSP is worthwhile for 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 to be verified, information which can be lost in the translation into the formalism. For instance, the removal of derived value constraints proposed in Section "Trivially Satisfiable Constraints" would not be possible without precise information about the property to be checked.

IX. Conclusions and Future Work

This paper presents a novel 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 analysed independently. Then, the satisfiability of the original model can be established by checking if at least one submodel (weakly satisfiable) or all submodels (strongly 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. Experimental results show that slicing can produce a significant speed-up in 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 believe that slicing is a useful addition to any UML/OCL satisfiability-checking toolkit. Furthermore, we have demonstrated the slicing technique on a real-world case study (DBLP conceptual schema) to analyse the benefits. This real-world case study is programmed in Alloy, which is a popular tool and widely used for verification of models. We applied the slicing technique and achieved drastic speed-up in this tool as well.

As regards our future work, we plan to create a verification engine with slicing techniques that can break unverifiable models into several independent submodels and verify them. With the help of slicing procedures, current verification methods will be efficient enough to verify models with an additional level of complexity that no current tool can handle.

ACKNOWLEDGEMENTS

This paper is a revised and extended version of the one presented at Automated Software Engineering (ASE 2010), Antwerp, Belgium had have received 38 citations.

REFERENCES

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *ACM/IEEE 10th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2007)*, volume 4735 of *LNCS*, pages 436–450, 2007.

[2] K. Androutsopoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, and Z. Li. Model projection: simplifying models in response to restricting the environment. In *ICSE*, pages 291–300, 2011.

[3] 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.

[4] P. Baumgartner, U. Furbach, M. Gross-Hardt, and T. Kleemann. Model based deduction for database schema reasoning. In *KI 2004: Advances in Artificial Intelligence*, pages 168–182. Springer, 2004.

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

[6] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling model slicers. In *MoDELS*, pages 62–76, 2011.

[7] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Kompren: Modeling and generating model slicers. *Software and Systems Modeling (SoSyM)*, 2012.

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

[9] 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.

[10] J. Cabot, R. Clarisó, and D. Riera. Verification of uml/ocl class diagrams using constraint programming. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 73–80, Washington, DC, USA, 2008. IEEE Computer Society.

[11] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC*, pages 1284–1291, 2012.

[12] D. Clark. Amorphous slicing for EFSMs. In *PLID' 07*, 2007.

[13] T. Clark. A general model-based slicing framework. In *Proc. of the Workshop on Composition and Evolution of Model Transformations*, 2011.

[14] 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.

[15] DBLP. Digital bibliography andy library project, 2012. http://guifre.lsi.upc.edu/DBLP.pdf.

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

[17] 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.

[18] 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.

[19] M. P. E. Heimdahl, J. M. Thompson, and M. W. Whalen. On the effectiveness of slicing hierarchical state machines: A case study. In *EUROMICRO*, pages 10435–10444, 1998.

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

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

[22] E. K. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen. Components, platforms and possibilities: towards generic automation for mda. In *EMSOFT*, pages 39–48, 2010.

[23] W. Ji, D. Wei, and Q. Zhi-Chang. Slicing hierarchical automata for model checking uml statecharts. In *Formal Methods and Software Engineering*, volume 2495 of *Lecture Notes in Computer Science*, pages 435–446. Springer Berlin / Heidelberg, 2002.

[24] 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.

[25] S. Khurshid. *Generating structurally complex tests from declarative constraints*. PhD thesis, Massachusetts Institute of Technology, 2003.

[26] 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.

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

[28] B. Korel, I. Singh, L. Tahat, and B. Vaysburg. Slicing of state-based models. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 34–. IEEE Computer Society, 2003.

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

[30] J. T. Lallchandani and R. Mall. A dynamic slicing technique for uml architectural models. *IEEE Trans. Software Eng.*, 37(6):737–771, 2011.

[31] K. Lano. Slicing of uml state machines. In *Proceedings of the* 9^{th} *WSEAS International Conference on Applied Informatics and Communications*, pages 63–69. World Scientific and Engineering Academy and Society (WSEAS), 2009.

[32] K. Lano and S. K. Rahimi. Slicing of uml models using model transformations. In *MoDELS (2)*, pages 228–242, 2010.

[33] K. Lano and S. K. Rahimi. Slicing techniques for uml models. *Journal of Object Technology*, 10:11: 1–49, 2011.

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

[35] 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.

[36] 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.

[37] 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.

[38] A. Shaikh, R. Clarisó, U. K. Wiil, and N. Memon. Verification-driven slicing of uml/ocl models. In *ASE*, pages 185–194, 2010.

[39] A. Shaikh and U. K. Wiil. UMLtoCSP (UOST): a tool for efficient verification of UML/OCL class diagrams through model slicing. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*

(FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012, page 37, 2012.

[40] A. Shaikh, U. K. Wiil, and N. Memon. UOST: UML/OCL aggressive slicing technique for efficient verification of models. In *System Analysis and Modeling: About Models - 6th International Workshop, SAM 2010, Oslo, Norway, October 4-5, 2010, Revised Selected Papers*, pages 173–192, 2010.

[41] A. Shaikh, U. K. Wiil, and N. Memon. Evaluation of tools and slicing techniques for efficient verification of uml/ocl class diagrams. *Adv. Software Engineering*, 2011, 2011.

[42] R. Singh and V. Arora. A practical approach for model based slicing. *IOSR Journal of Computer Engineering*, 12(4):18–26, 2013.

[43] 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.

[44] W. Sun, B. Combemale, and R. B. France. Towards the use of slicing techniques for an efficient invariant checking. In *Companion Proceedings of the 14th International Conference on Modularity*, pages 23–24. ACM, 2015.

[45] 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.

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

[47] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[48] Z. Ujhelyi, Á. Horváth, and D. Varró. Dynamic backward slicing of model transformations. In *ICST*, pages 1–10, 2012.

[49] E. Uzuncaova. *Efficient specification-based testing using incremental techniques*. ProQuest, 2008.

[50] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. Testing software product lines using incremental test generation. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 249–258. IEEE, 2008.

[51] E. Uzuncaova and S. Khurshid. Program slicing for declarative models. *ACM SIGSOFT Software Engineering Notes*, 31(6):1–2, 2006.

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

[53] E. Uzuncaova and S. Khurshid. Constraint prioritization for efficient analysis of declarative models. In *FM 2008: Formal Methods*, pages 310–325. Springer, 2008.

[54] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *Software Engineering, IEEE Transactions on*, 36(3):309–322, 2010.

[55] S. Van Langenhove and A. Hoogewijs. sv_tl: System verification through logic tool support for verifying sliced hierarchical statecharts. In *Recent Trends in Algebraic Development Techniques*, volume 4409 of *Lecture Notes in Computer Science*, pages 142–155. Springer Berlin / Heidelberg, 2007.

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

[57] J. Zhao. Applying slicing technique to software architectures. In *ICECCS*, pages 87–99, 1998.