SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2211

A feedback technique for unsatisfiable UML/OCL class diagrams

Asadullah Shaikh[1,2,*,†] and Uffe Kock Wiil[1]

[1]*The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Denmark*
[2]*Department of CS & IT, Institute of Business & Technology, Pakistan*

SUMMARY

In Model-Driven Development (MDD), detection of model defects is necessary for correct model transformations. Formal verification tools and techniques can to some extent verify models. However, scalability is a serious issue in relation to verification of complex UML/OCL class diagrams. We have proposed a model slicing technique that slices the original model into submodels to address the scalability issue. A submodel can be detected as unsatisfiable if there are no valid values for one or more attributes of an object in the diagram or if the submodel provides inconsistent conditions on the number of objects of a given type. In this paper, we propose a novel feedback technique through model slicing that detects unsatisfiable submodels and their integrity constraints among the complex hierarchy of an entire UML/OCL class diagram. The software developers can therefore focus their revision efforts on the incorrect submodels while ignoring the rest of the model. Copyright © 2013 John Wiley & Sons, Ltd.

Received 7 May 2012; Revised 8 May 2013; Accepted 9 May 2013

KEY WORDS: model slicing; feedback technique and slicing UML/OCL models with feedback

1. INTRODUCTION

In the context of Model Driven Development (MDD), Unified Modeling Language/Object Constraint Language (UML/OCL) class diagrams are used as higher level designs of software systems. Afterwards, these class diagrams are transformed into software code to speed up the software development process. With the help of this kind of transformation, developers can save their time and effort by using the transformed code to make their software product.

Considering the fact that within MDD, UML/OCL class diagrams play an important role for model analysis, design, and transformation; therefore, the verification of these UML/OCL class diagrams at earlier stages is an essential task to check the correctness of the model properties, that is, verification of UML/OCL class diagram with several OCL integrity constraints. We have considered the static structure diagram that describes the structure of a system modeled as a UML class diagram. Complex integrity constraints will be expressed in OCL. In this context, the fundamental correctness properties of a model are *satisfiability* and *unsatisfiability*.

> "Two different notions of satisfiability can be checked - either weak satisfiability or strong satisfiability. A class diagram is weakly satisfiable if it is possible to create a legal instance/object of a class diagram that is non-empty, that is, it contains at least one object from some class. Alternatively, strong satisfiability is a more restrictive condition requiring that the legal instance has at least one object from each class and a link from each association" [1–4].

*Correspondence to: Asadullah Shaikh, Department of CS & IT, Institute of Business & Technology (Biztek), Pakistan.
[†]E-mail: shaikhasad@hotmail.com and ashaikh@ibt.edu.pk

The unsatisfiability may occur if there are no valid values for one or more attributes of an object in the diagram.

> "Apart from these notions of satisfiability, there are few other notions such as *liveliness of class, lack of constraint subsumptions* and *lack of constraint redundancies*. In liveliness of class, the model must have finite instances where the population of class must be non-empty. In lack of constraint subsumptions, if there are two integrity constraints C_1 and C_2, the model should have finite instances where C_1 is satisfiable but C_2 is not; otherwise, C_1 *subsumes* C_2. In lack of constraint redundancies, if there are two integrity constraints C_1 and C_2, the model should have finite instances where only one constraint is satisfied. In other cases, C_1 and C_2 are redundant" [5].

There are formal verification tools for automatically checking correctness properties on models [6–9], but the lack of scalability and proper feedback in case of unsatisfiable models is usually a drawback. The UML/OCL class diagram is considered as satisfiable if the interaction of all its integrity constraints is satisfiable. Alternatively, if any integrity constraint violates a condition given in an OCL expression, then the class diagram is unsatisfiable. When the class diagram is unsatisfiable, the main concern for the designers is to correct it. Therefore, the need for proper feedback is ever present to make the necessary corrections to change the unsatisfiable class diagram into a satisfiable class diagram. Current approaches to this problem identify the failed class diagram in general, that is, if the interaction of any single constraint is unsatisfiable then the entire model is unsatisfiable, but it does not highlight the exact position of the problem (e.g., the exact OCL constraint(s) that made the entire model unsatisfiable). It creates difficulty for the developers, if the UML/OCL class diagram is complex having a number of classes, associations, attributes, relationships, and OCL invariants and one or more properties are unsatisfiable. In this case, the designers need to check the expression of each constraint to find out the failing properties and correct them one by one.

We have addressed the general problem related to lack of scalability in formal verification tools by proposing a model slicing technique for disjoint sets of submodels and non-disjoint sets of submodels [2–4]. However, in this paper, we address another problem concerning feedback if one or more submodels are unsatisfiable. The proposed slicing technique accepts a UML class diagram annotated with OCL constraints as an input, breaks the original model m into m_1, m_2, m_3,…,m_n submodels while abstracting unnecessary model elements. The process ensures that the model m is satisfiable if all m_1, m_2, m_3,…,m_n submodels are satisfiable. Then, satisfiability of each slice is checked independently, and the results are combined to assess the satisfiability of the entire model. The proposed slicing technique improved the efficiency of the verification process for large and complex UML/OCL models. Therefore, with the help of the slicing procedure UML/OCL class diagrams that were not verifiable before are now verifiable. For example, the class diagram (Tracking System) has 50 classes, 60 associations, 72 attributes, and 5 integrity constraints. Before applying the slicing technique, UMLtoCSP tool [7] verified the model 'tracking system' in 3605.35 s (60.08 min). However, after implementing the slicing algorithm in UMLtoCSP(UOST) [10], the tool verified the same model in 0.031 s. We achieved a 99% speedup in the verification of the tracking system class diagram and 43% speedup for a real world UML/OCL class diagram, that is, DBLP conceptual schema [2, 4].

After applying the slicing technique, if the output of the tool is unsatisfiable then one or more constraints are unsatisfiable. This indicates that some submodel having an expression in OCL is violated and that the objects of the classes cannot be instantiated according to the given OCL expression. Any submodel of an original model can be unsatisfiable if there are no valid values for one or more attributes of an object in the diagram or the model provides inconsistent conditions on the number of objects of a given type.

In this paper, we propose a feedback technique that detects one or more specific unsatisfiable submodel(s) with its integrity constraints among the complex hierarchy of an entire UML/OCL class diagram. Our proposal is based on the following: (1) application of the slicing technique; (2) detection of unsatisfiable submodel(s); and (3) identification of the invariant(s) that cause(s) the unsatisfiability. The feedback technique will help software developers to highlight unsatisfiable constraints with minimal effort. The proposed feedback technique can be used for the detection

DOI: 10.1002/spe

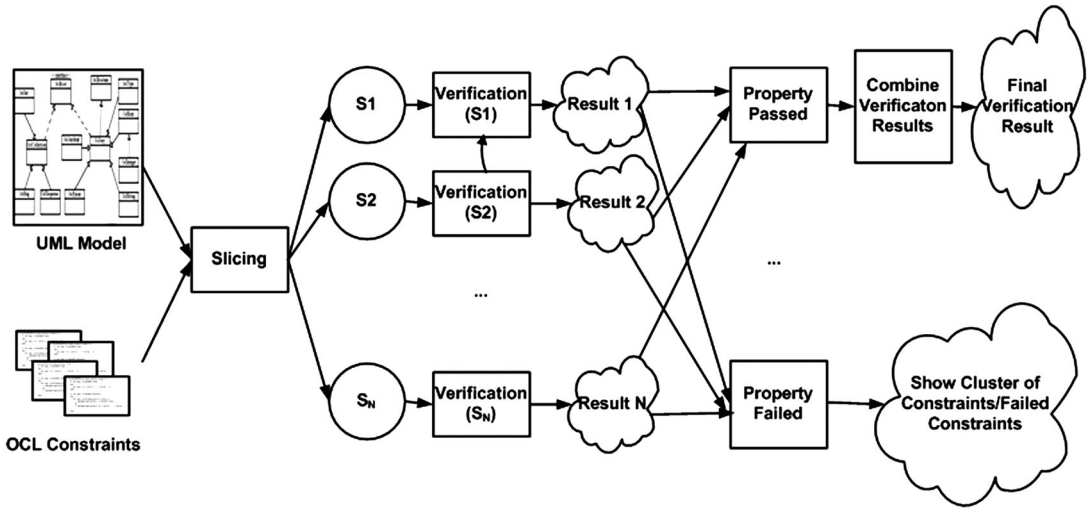


Figure 1. Model slicing and feedback process.

of any unsatisfiable UML/OCL class diagram. Figure 1 represents the concept of model slicing with feedback.

The remainder of the paper provides an overview of the slicing technique, and describes and evaluates the feedback technique. Section 2 explores detection of unsatisfiable class diagrams. Section 3 introduces the concept of model slicing. In Section 4 the analysis of OCL constraints is discussed whereas Section 5 presents the concept of graph-based representation. Section 6 focuses on the proposed technique along with a running example and explains the detection of unsatisfiable models. In Section 7, related work is presented. Finally, Section 8 provides the conclusions and identifies directions for future work.

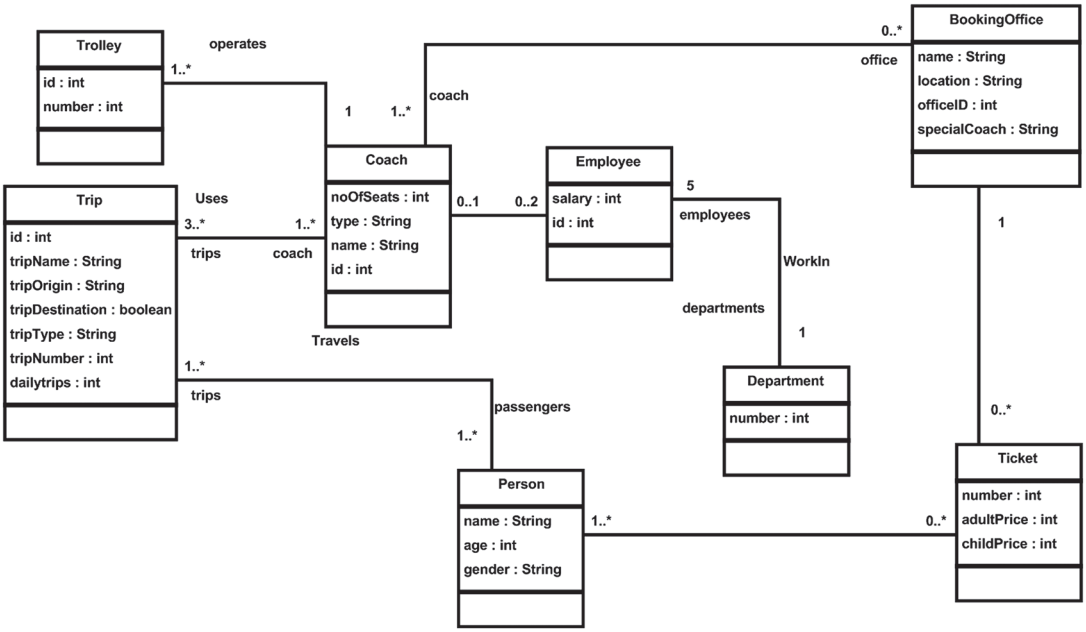
2. DETECTION OF UNSATISFIABLE CLASS DIAGRAMS

Unfortunately, there are currently few tools and techniques that support verification of UML/OCL class diagrams and, more precisely, these tools can only verify UML models especially UML/OCL class diagrams up to a limited complexity, that is, the lack of scalability is usually a drawback in current verification tools. Another problem with these tools is the lack of feedback if the UML/OCL class diagram is unsatisfiable. Figure 2 introduces the 'model coach' class diagram in which one or more integrity constraints are unsatisfiable.

We verified the model coach UML/OCL class diagram in a tool called UMLtoCSP [7] and the given output of the tool is 'No satisfying instance can be found within the specified search space'. It indicates that one or more invariants are unsatisfiable, but the tool failed to identify the specific unsatisfiable constraints. In this case, the developers need to check and correct all invariants that is a time consuming task especially when the model is complex, having hundreds of thousands of classes and invariants.

Considering the feedback issue in an external tool, we programmed a more complex real world class diagram of the DBLP conceptual schema [11] in UML2Alloy [12]. The class diagram of the DBLP conceptual schema is based on 17 classes and 27 integrity constraints. The overall interaction of the class diagram is unsatisfiable due to one or more expressions given in integrity constraints, that is, no valid instances can be generated. UML2Alloy transforms the model into Alloy language and Alloy Analyzer translates the model into a boolean expression that is analyzed by SAT solvers. If there are no valid instances found, the UnSat Core [13] will highlight the relevant portions of the original model that contributed to the unsatisfiability. However, the designers need to correct these UnSat cores one by one, because it is difficult to find out multiple UnSat cores at a time.

DOI: 10.1002/spe



context Coach **inv** MinCoachSize:
self.noOfSeats $\geq$ 10

context Coach **inv** MaxCoachSize:
self.trips −>forAll(t | t.passengers −>size() $\leq$ noOfSeats)

context Department **inv** EmployeeSize:
self.employee()−>size() = 7

context Department **inv** DeptartmentSize:
Department::allInstances() −>size() = 1

context Ticket **inv** PositiveTicketNumber:
Ticket::allInstances() −>forAll(a | a.number$\geq$ 1)

context BookingOffice **inv** UniqueOfficeID:
BookingOffice::allInstances() −>isUnique (t | t.officeID)

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

3. THE CONCEPT OF MODEL SLICING

We have used the model slicing approach to detect unsatisfiable submodel(s). The slicing algorithms automatically breaks the UML/OCL model into several independent submodels and abstracts the unused components of class models. The proposed method takes a UML class diagram annotated with OCL invariants as an input. Figure 2 introduces the class diagram that will be used as an example for the slicing and feedback technique; it models the information system of a bus company.

Our goal is to detect whether the UML/OCL class diagram is strongly satisfiable, that is, whether it is possible to generate legal instances from each class and a link for each association. The output will be either 'satisfiable' or 'unsatisfiable'. If the class diagram is unsatisfiable then there exists one or more constraints whose interaction is unsatisfiable.

The slicing procedure breaks the UML/OCL class diagram into sets of disjoint and non-disjoint slices where each slice is a subset of the original model. Afterwards, each slice is verified separately by the verification engine, and the result of whole model is obtained by combining the results of all slices. In case of strong satisfiability, it is important to ensure whether all slices are strongly

DOI: 10.1002/spe

satisfiable. In case of weak satisfiability, it is enough to check that at least one slice is weakly satisfiable.

On the contrary, it is also essential to check if the original model is unsatisfiable, the final output will also be unsatisfiable. For example, in strong satisfiability, there should be at least one slice whose interaction will be strongly unsatisfiable, and in case of weak satisfiability, all slices should be weakly unsatisfiable. The formal proof of this procedure can be found in [14].

The slicing procedure partitions the UML/OCL class diagram into sets of disjoint and non-disjoint submodels. The selection of disjoint and non-disjoint submodels depends upon the user. We have a detailed explanation of the disjoint slicing technique in [2] and the non-disjoint slicing technique in [4].

4. STUDY OF OBJECT CONSTRAINT LANGUAGE CONSTRAINTS

Object constraint language is the language for describing the rules that apply on UML models. With the help of OCL, it is possible to define expressions on UML models. These expressions can either be true or false. In this section, we focus on analysis of OCL constraints to remove information irrelevant to the model's satisfiability. We have introduced the concept of constraint support and trivially satisfiable patterns. Constraint support helps to identify model elements that are constrained by an invariant and therefore, it is possible to analyze two or more constraints belonging to the same model elements. Trivially satisfiable patterns help to remove those constraints that do no affect satisfiability.

4.1. Constraint support

The constraint support of OCL invariants represents the set of classes restricted by those constraints. It identifies classes that are used in the same constraints and therefore must be grouped and analyzed within the scope of the same slice. We identify OCL invariants and group them if they restrict the same model elements. We call this *clustering of constraints* (constraint support). Algorithm 1 illustrates the computation of clusters.

The constraint support defines the scope of a constraint. The support information can be used to partition a set of OCL invariants into a set of independent clusters of constraints, where each cluster can be verified separately. The following procedure 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 support (i.e., $\text{support}(x) \cap \text{support}(y) \neq \emptyset$) 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.

By using the information from Table I, we can identify four clusters in our model: invariants MinCoachSize and MaxCoachSize (support: Coach, Trip, and Person); invariants EmployeeSize and DepartmentSize (support: Department, and Employee); invariant PositiveTicketNumber (support: Ticket, BookingOffice, Coach, Trip, and Person); and invariant UniqueOfficeID (support: BookingOffice, Coach, Trip, Person, and Ticket). In the following section, we describe additional analysis that can abstract clusters of constraints.

4.2. Trivially satisfiable invariants

The detection of trivially satisfiable invariants can improve the efficiency of the verification process by eliminating those OCL constraints that do not affect satisfiability. It is hard to detect trivially satisfiable constraints, therefore, we limit ourselves to particular patterns. We have defined a couple of patterns that can be safely removed from the problem. The first pattern is the *key constraint* stating that a given attribute must be unique, for example, Type::allInstances() –> isUnique (object|object.attribute). If there is an attribute of type integer, float, or string and it is not used by any other constraint, it can be trivially satisfied.

DOI: 10.1002/spe

Table I. Support, attributes, and navigations in the running example.

Invariant	Support	Attributes	Navigations
MinCoachSize	Coach	Coach.noOfSeats	None
MaxCoachSize	Coach, Trip, Person	Coach.noOfSeats	Travels, Uses
EmployeeSize	Department, Employee	None	WorkIn
DepartmentSize	Department	None	None
PositiveTicketNumber	Ticket	Ticket.number	None
UniqueOfficeID	BookingOffice	BookingOffice.OfficeID	None

Table II. Patterns with conditions [2].

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.

The second pattern is *derived value constraint*. In this type of pattern the attributes depend on the values of each other. For example, *self.attribute op expression* where attribute can be type (boolean, integer, float, and string) not referenced by any other constraint, *op* is a relational operator, and expression is an OCL expression.

The *key constraint* and *derived value constraint* cannot make the model unsatisfiable because the values of attributes do not depend on each other. If the attribute is referenced twice in the OCL expression then there is a possibility of unsatisfiability. Table II defines the patterns with trivially satisfiable constraints with their conditions [2].

In the beginning, each constraint is trivially satisfiable unless the attribute of that constraint is not referenced by any other constraint. Considering our running example, MinCoachSize is a derived value constraint but invariant MaxCoachSize restricts the same attribute 'noOfSeats' that is already referenced by MaxCoachSize . Therefore, invariants MinCoachSize and MaxCoachSize will not be considered as trivially satisfiable because they are referencing the same attribute. The other type of constraints, foe example, Type::allInstances() –>size() = x, if the 'type' is referenced more than once, it will not be counted as a trivially satisfiable pattern. For example, invariants EmployeeSize and DeptartmentSize are referencing the same type. Finally, invariant PositiveTicketNumber and UniqueOfficeID are key constraints that can be abstracted.

5. GRAPH BASED REPRESENTATION

A graph-based representation called *dependency graph* is used to capture the dependencies among classes in a class diagram. The computation of partitions will simply consist of computing the connected components of the graph.

A dependency graph is an undirected graph where each vertex is a class. The relationships are defined through a graph-based representation called *flow graph*. More formally, a flow graph is a labeled directed pseudograph, that is, adjacent vertices are connected by arcs (directed) instead of edges (undirected); there can be arcs from one vertex to itself and multiple arcs between two vertices. Each vertex of the flow graph is a class of the class diagram and each arc from X to Y captures a dependency between X and Y. Dependency arcs are labeled with information about the *type of dependency*.

DOI: 10.1002/spe

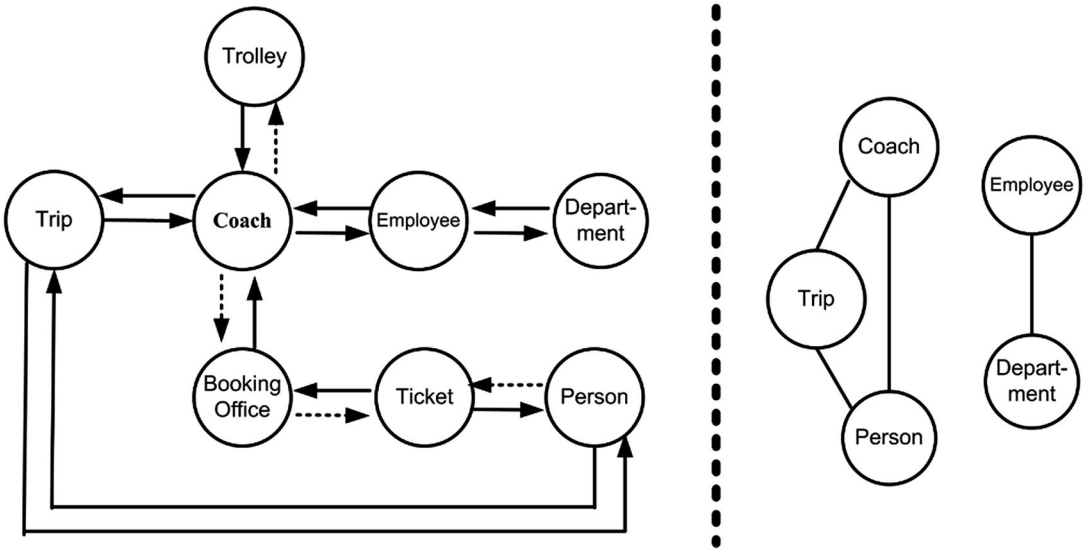


Figure 3. Flow graph (left) and dependency graph (right) for the running example.

Two types of relationships among classes are considered: tightly associated and loosely associated classes. An association with a lower bound of 0 (e.g., 0..3) is called loosely coupled; this means if an object of class A is instantiated, then it is not necessary that an object of class B must be instantiated. However, tightly coupled classes are the opposite of loosely coupled classes, i.e., they have an association with a lower bound greater than 1 (e.g., 1..*). Arcs with label 0 can be removed from the graph because they do not impose any constraint.

By using the previous information, the dependency graph can be created in two steps: (1) identification of classes, that is, classes constrained by OCL invariants; and (2) we iteratively add classes that are tightly coupled. For example, Figure 3 shows the flow graph for the running example, where the visual styles of arcs depict different types of dependencies. Similar representations are also used in other slicing approaches [15–17].

6. OVERVIEW OF FEEDBACK TECHNIQUE

The goal of the feedback technique is to determine the specific unsatisfiable submodel(s) with its OCL invariants. The following section focuses on the entire process of the proposed method: (1) remove trivially satisfiable constraints; (2) compute slices; (3) detect unsatisfiable submodels; and (4) identify OCL invariants whose interaction is unsatisfiable.

6.1. Feedback technique by example

Algorithm 1 describes the step-by-step slicing procedure for the detection of unsatisfiable submodels with its invariants. First, before applying the slicing procedure, we remove the trivially satisfiable constraints, that is, invariants PositiveTicketNumber (support: Ticket, BookingOffice, Coach, Trip, and Person); and invariant UniqueOfficeID (support: BookingOffice, Coach, Trip, Person, and Ticket) as observed in Section 4.2. After the removal of trivially satisfiable constraints, we identify clusters of constraints by using the information given in Section 4.1. Originally, there were four clusters in the model but a couple of clusters were removed because of detection of trivially satisfiable constraints. The remaining clusters are invariants MinCoachSize and MaxCoachSize (support: Coach, Trip, and Person) and invariants EmployeeSize and DepartmentSize (support: Department and Employee).

Second, the slicing procedure is applied over the running example (Figure 2). We will receive two submodels: Coach, Trip, Person (submodel 1) and Department, and Employee (submodel 2). Figure 4 highlights the final slices passed to the verification tool UMLtoCSP(UOST) [10] for strong satisfiability whereas the rest of the classes, associations, and attributes will be removed from the problem as they do not affect satisfiability.

DOI: 10.1002/spe

Algorithm 1 Slicing with the identification of failed constraints

Input: Property being verified

Output: A partition P of the model M into non-necessarily disjoint submodels

```
1: G ← BuildFlowGraph(M) {Creating the flowgraph}
2: for each constraint c_1, c_2, ...., c_n in M do
3:    remove trivially satisfiable constraints.
4: end for
5: {Cluster the OCL constraints}
6: for each pair of constraints c1, c2 in M do
7:    if ConstraintSupport(M, c1) ∩ ConstraintSupport(M, c2) ≠ ∅ then
8:       MergeInSameCluster(c1, c2)
9:    end if
10: end for
11: {Work on each cluster of constraints separately}
12: for each cluster of constraints Cl do
13:    subModel ← empty model {Initialize the subModel to be empty}
14:    {Initialize worklist}
15:    workList ← Union of the ConstraintSupport of all constraints in the cluster
16:    while workList not empty do
17:       node ← first(workList) {Take first element from workList and remove it}
18:       workList ← workList \ node
19:       for each subclass or superclass c of node do
20:          subModel ← subModel ∪{c}
21:          if c was not before in the subModel then
22:             workList ← workList ∪{c}
23:          end if
24:       end for
25:       for each class c tightly coupled to node do
26:          if Property = weak SAT then
27:             subModel ← subModel ∪{c}
28:          else if Property = strong SAT then
29:             workList ← workList ∪{c}
30:          end if
31:       end for
32:    end while
33: end for
34: for each subModel in Model M do
35:    if Property (weak SAT or strong SAT ) = True then
36:       return subModel ← Object Diagram
37:    else
38:       return failed subModel
39:       for each inv_i in subModel do
40:          if inv_i violates SAT property then
41:             return inv_i
42:          end if
43:       end for
44:    end if
45: end for
```


DOI: 10.1002/spe

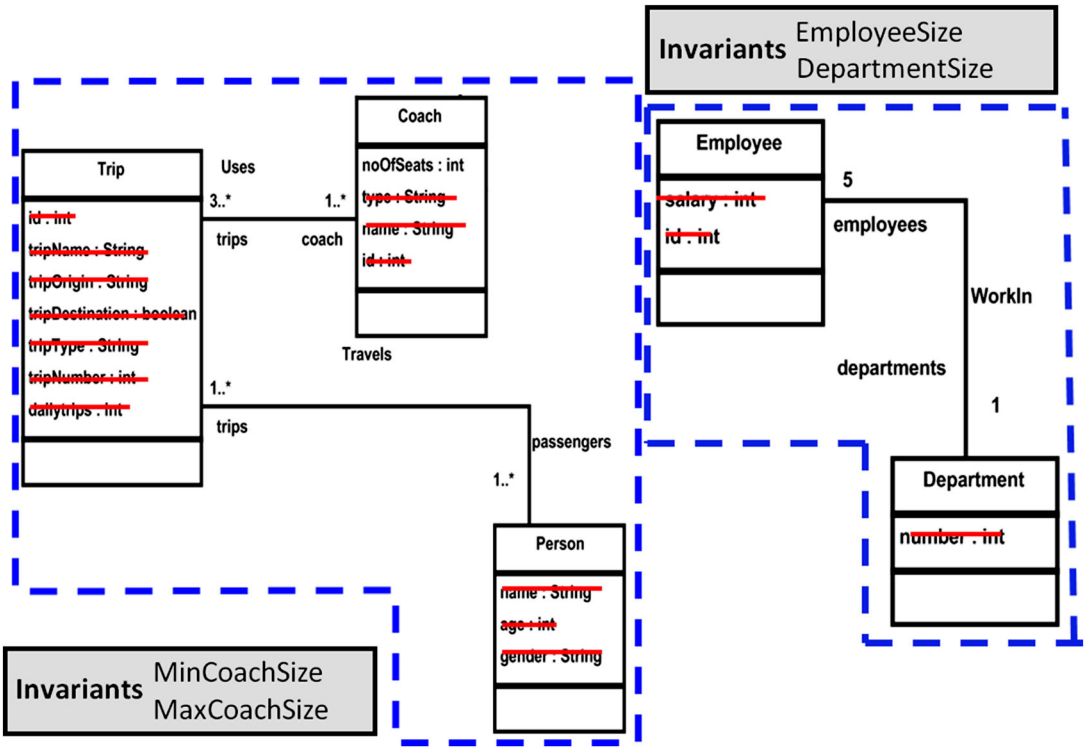


Figure 4. Submodels for the verification of strong satisfiability.

Third, the satisfiability of each submodel is checked independently, that is, the interaction of submodel 1 is satisfiable; however, submodel 2 is unsatisfiable due to violation of one or more properties. The proposed solution detects the specific unsatisfiable submodel by analyzing the interaction of each constraint with the help of constraint satisfaction problem (CSP) [18].

Finally, the algorithm of the feedback technique is implemented in the UMLtoCSP(UOST) tool, therefore, CSP formalism detects the failed property by evaluating the expression in the form of true or false. In this case, the output of the tool will be unsatisfiable as one of the submodels is unsatisfiable, that is, submodel 2. Figure 5 presents the output of the tool for satisfiable model (submodel 1) and unsatisfiable model (submodel 2) with their respective OCL constraints.

Submodel 1 restricts two constraints whose interaction is satisfiable; however, submodel 2 also restricts two constraints but one of the constraint's interaction is unsatisfiable; therefore, the overall interaction of the model is unsatisfiable. The proposed approach will help developers to focus their attention on incorrect submodels while ignoring the rest of the model.

6.2. *Detection of unsatisfiable submodels*

Any submodel can be unsatisfiable due to several reasons. First, it is possible that one of the invariants provides inconsistent conditions on the number of objects of a given type. For example, 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. So it seems that an unsatisfiable model either contains an unsatisfiable textual or graphical constraint. In our running example, the cause of unsatisfiability is multiplicities of association ends. By using the information given in submodel 2, our proposed algorithm considers the failed model and its specific invariants with the help of CSP, that is, invariants EmployeeSize and DepartmentSize (support: Department and Employee) to detect the specific property that causes unsatisfiability. The given expression (Department::allInstances() –>size() = 1) for invariant 'DeptartmentSize' satisfies the condition, therefore, the possible source of unsatisfiability is the next invariant 'EmployeeSize', that is, self.employee() –>size() = 7. This invariant violates the given expression, i.e., the size of

DOI: 10.1002/spe

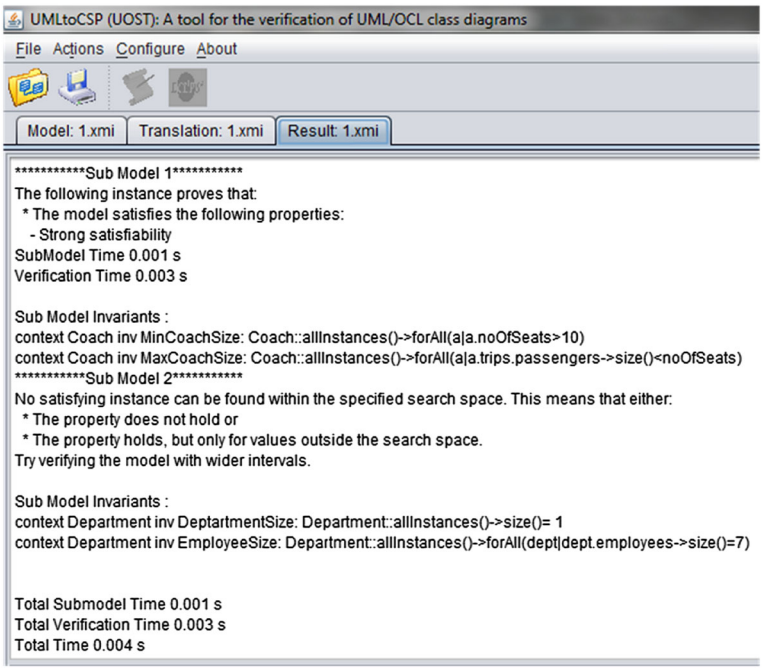


Figure 5. Screenshot for submodel 1 and submodel 2.

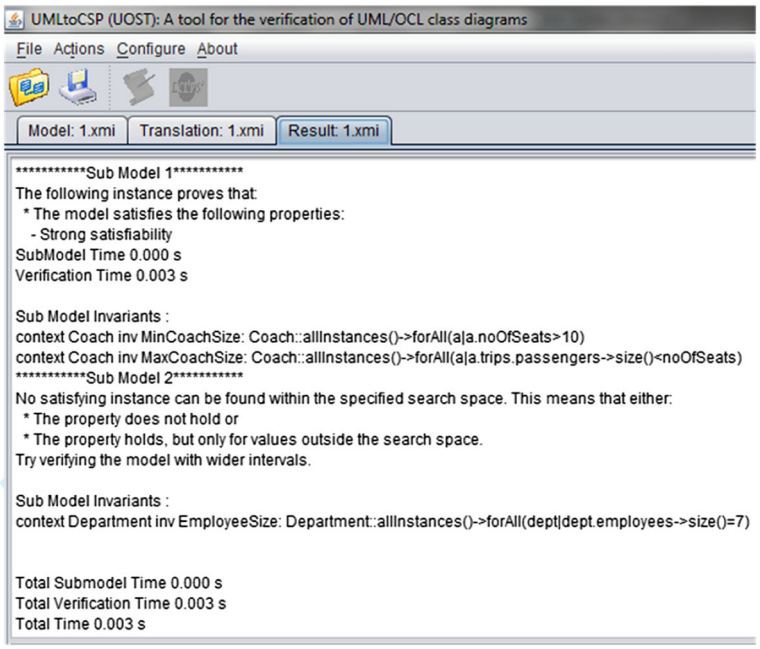


Figure 6. Specific unsatisfiable invariant of submodel 2.

employees in the department must be equal to 7, however, considering the associations in the class Employee, the maximum number of employees in the department could be 5. This detection process will be continued for all invariants of failed submodel(s). As an example, Figure 6 illustrates a specific invariant that makes the entire class diagram unsatisfiable [‡].

[‡]There is an option in the tool where a developer can view the total number of invariants belonging to a specific submodel, that is, satisfiable constraints and unsatisfiable constraints or the specific unsatisfiable constraints.

DOI: 10.1002/spe

Figure 7. Tool output in case of more than one unsatisfiable invariants.

The feedback technique is able to detect more than one unsatisfiable invariant. For example, if we add one more unsatisfiable invariant that is as follows:

```
context Department
inv DeptartmentSize: Department::allInstances()->size() = 9
```

UMLtoCSP(UOST) will automatically highlight unsatisfiable invariants as shown in Figure 7. The developers can therefore correct the highlighted invariants while ignoring other satisfiable constraints.

These unsatisfiable invariants can be corrected by changing the values of multiplicities either textually or graphically because the cause of unsatisfiability is multiplicities of association ends. The interaction of invariant `DepartmentSize` can be satisfiable by changing the value from `size()= 9` to `size()= 1` and invariant `EmployeeSize` from `size()= 7` to `size()= 5`. Figure 8 shows the output of the tool when both invariants are satisfiable, and therefore, the entire model is satisfiable[§].

[§]UMLtoCSP(UOST) generated three submodels after correcting invariant DepartmentSize and EmployeeSize. We have shown only one submodel in Figure 8 as an example.

DOI: 10.1002/spe

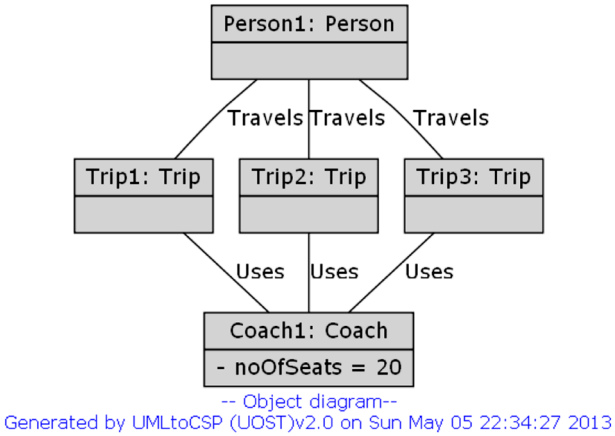


Figure 8. Tool output after correction of unsatisfiable invariants.

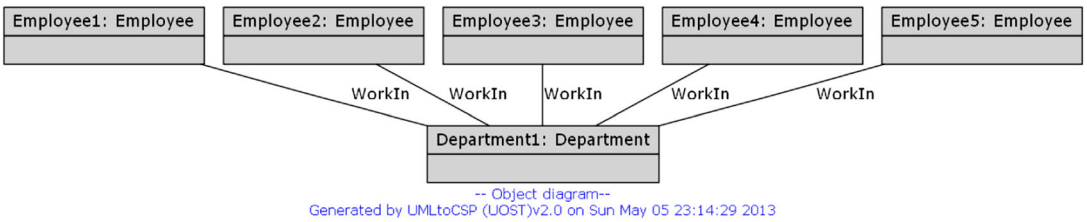


Figure 9. Tool output in case of weak satisfiability.

UMLtoCSP(UOST) has several options to show the output of the tool other than strong satisfiability [10]. Therefore, the user can check if the model is weakly satisfiable. Figure 9 shows the output of the tool in case of weak satisfiability.

6.3. Experimental results

In this section, we explore the experimental results regarding how fast we can compute the unsatisfiable submodel(s) and generate feedback. For these experiments, we compare the verification time of several UML/OCL class diagrams by using the original UMLtoCSP tool [7] and UMLtoCSP(UOST) with slicing and feedback [10]. Table III describes the examples used for our comparison. In each example, the property being verified is strong satisfiability and the interaction of each class diagram is unsatisfiable. The examples 'scripts 1–3' were programmatically generated to test large unsatisfiable models.

Table IV shows the experimental results computed using an Intel Core 2 Duo Processor 2.1 GHz with 2 GB of RAM. All times are measured in seconds and a time-out limit has been set at 2 h (7200 s). For each example, we give the original verification time, the number of slices in which the model is divided, the time required to perform all the UML/OCL slicing analysis, and the verification time after the slicing to generate the feedback for unsatisfiable submodel(s).

The experimental results show that the proposed slicing and feedback techniques are very efficient even in diagrams with hundreds of classes. With small unsatisfiable models, UMLtoCSP already performed well, but the feedback in the output screen is not illuminating. However, with the help of slicing, it is possible to get instant results whether the model is satisfiable or unsatisfiable. In case of an unsatisfiable model, the feedback technique specifies the specific invariants that cause the unsatisfiability.

DOI: 10.1002/spe

Table III. Description of the UML/object constrained language examples.

Example	Classes	Associations	Invariants	Strongly satisfiable?
Paper-researcher	2	2	5	No
Shop model	5	4	5	No
Coach	8	8	6	No
DBLP conceptual schema	17	25	26	No
Script 1	100	53	2	No
Script 2	500	227	5	No
Script 3	1000	505	5	No

Table IV. Description of experimental results.

Example	OVT	Slices	ST (s)	SVT (s)	Speedup (%)
Paper-Researcher	0.001 s	1	0.00	0.001	0
Shop Model	4035.21 s	3	0.00	0.003	99.99
Coach	5008.76 s	2	0.01	0.003	99.99
DBLP Conceptual Schema	Time-out	2	0.02	0.05	99.99
Script 1	Time-out	2	0.02	0.05	99.99
Script 2	Time-out	4	0.09	0.07	99.99
Script 3	Time-out	4	0.29	0.34	99.99

OVT, original verification time; SVT, total verification time for all slices; ST, slicing time.

7. RELATED WORK

There are two main sources of previous work relating to feedback for unsatisfiable models: (1) find out the property that may violate an integrity constraint; and (2) unsatisfiable cores. Cabot *et al.* [19] proposed an approach that automatically determines the conditions that may violate an integrity constraint. Before transforming the model into code, determination of the properties that may violate an integrity constraint is essential.

There are other approaches for detecting unsatisfiable cores. Using unsatisfiable cores for debugging design errors has been proposed by Andre *et al.* [20]. The proposed method is based on a debugging framework that identifies unsatisfiable cores. The solution is similar to SAT-based debugging. With the help of this method, it is possible to detect faults at earlier stages and speed up the debugging process. Similar work is proposed by Torlak *et al.* [13] for detection of unsatisfiable cores of declarative specifications. This work for finding minimal unsatisfiable cores of declarative specification is based on recycling core extraction. The SAT solver is the main source for highlighting the UnSat cores. The computation of UnSat cores is a time consuming process, therefore, efficient compilation methods are used to generate fast results. Initially, the core is returned by the SAT solver, which is later on translated back into high-level specification language.

Our work is different from previous work: if the model is complex, it is difficult to verify and find out unsatisfiable OCL constraints. Our proposed method can detect an unsatisfiable submodel and its integrity constraints for both simple and complex class diagram. We address this problem with the help of a slicing procedure.

Another source related to our feedback approach is model slicing. Model slicing is used to keep necessary parts of the model while ignoring the rest of the information. There are a few slicing approaches that break UML/OCL models (class diagram, sequence diagram, and activity diagram) into several chunks to transform them into code. These slicing approaches do not provide feedback if one or more submodels are unsatisfiable. Feedback is only provided to a certain extent, that is, detection of unsatisfiable submodels. However, what makes those submodels unsatisfiable is hardly addressed in current literature.

The theory of model slicing is invented to support and maintain large UML models. There are few methods that support automated extraction of a subset of the model. Viewing large UML models at first sight is quite a complex task. Context free slicing of the model summarizes static and structural

DOI: 10.1002/spe

characteristics of a UML model [21], where the UML model and the multi-graph captures the nodes and elements of information in the UML model and the relationship between them. A different approach focusing on class diagram comprehension is the use of coupling metrics [22] to slice large models for visualization. Finally, the slicing of models consisting of both UML class diagrams and UML sequence diagrams is considered in [15].

The most recent work on slicing of UML models using model transformation is presented by Kevin Lano [23]. The approach produces smaller models considering the properties of a model. The purpose of slicing is to break the model into several submodels for better analysis and understanding. The slicing technique is applied on UML class diagrams and state machines. The main criterion of slicing is model transformation.

Further directions for related work are OCL impact analysis techniques. Altenhofen *et al.* [24] presented an approach to improve model efficiency by considering the changes in the model to decrease the number of reevaluations. This approach is useful for efficient support of OCL in large-scale modeling environment. Further work on OCL impact analysis is presented by Uhl *et al.* [25] where the authors proposed an algorithm for OCL 2.2 expression and a model change that can efficiently examine several elements in which the values have been changed because of the event. The algorithm is efficient enough and can handle the optimal complexity for operation calls and recursive operation calls. Moreover, Goldschmid, and Uhl [26] proposed an approach for textual modeling. The technique is based on incremental process that helps to keep model elements for partial views in case of changes to textual view representation.

8. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a new technique to provide feedback for unsatisfiable UML/OCL class diagrams as the detection of failed properties in a complex class diagram is a difficult and time consuming task. To address this complex problem, we first slice the original class diagram into several independent submodels. Afterwards, the proposed technique automatically detects the unsatisfiable submodel(s) with its OCL constraints. Any submodel may have one or more failed constraints; therefore, in the second stage of the proposed method, we identify the exact failed properties/constraints that cause unsatisfiability. Experimental results show that the computation of unsatisfiable models is efficient. It will help developers to correct the specific parts of the model while ignoring the rest of the complex hierarchy. The feedback technique will help optimize certain complex tasks of software developers.

As our future work, we plan to explore two research directions. First, we plan to develop a more specific feedback technique that can suggest possible corrections for a failed property. For example, to provide an exact solution for a failed invariant. Secondly, we plan to develop a technique that automatically documents the changes in the model and its transformations. OCL and UML diagram modifications are important, especially when the model is unsatisfiable. Currently, developers make changes in the OCL file and save it without recording previous states. As a result, there is no log that tracks the changes in the file.

REFERENCES

1. Queralt A, Teniente E. Reasoning on UML Class Diagrams with OCL Constraints. In *ER'2006*, Vol. 4215, LNCS. Springer-Verlag, 2006; 497–512.
2. Shaikh A, Clarisó R, Wiil UK, Memon N. Verification-driven Slicing of UML/OCL Models. In *ASE*, 2010; 185–194.
3. Shaikh A, Wiil UK, Memon N. Evaluation of Tools and Slicing Techniques for Efficient Verification of UML/OCL Class Diagrams. *Adv. Software Engineering* 2011.
4. Shaikh A, Wiil UK, Memon N. UOST: UML/OCL Aggressive Slicing Technique For Efficient Verification of Models. In *6th International Workshop on System Analysis and Modelling (SAM 2010)*, Vol. 6598, *Lecture Notes in Computer Science*. Berlin Heidelberg, 2011; 173–192.
5. Cabot J, Clarisó R, Riera D. Verification of UML/OCL Class Diagrams Using Constraint Programming. In *ICSTW '08*. IEEE Computer Society, 2008; 73–80.
6. Brucker A, Wolff B. HOL-OCL: A Formal Proof Environment for UML /OCL. In *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, 2008; 97–100.

DOI: 10.1002/spe

7. Cabot J, Clarisó R, Riera, D. UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming. In *ASE'2007*. ACM, 2007; 547–548.
8. Gogolla M, Bohling J, Richters M. Validation of UML and OCL Models by Automatic Snapshot Generation. In *Proceedings of the 6th Int. Conf. Unified Modeling Language (UML 2003)*. Springer, 2003; 265–279.
9. Jackson D. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology* 2002; **11**(2):256–290.
10. Shaikh A, Wiil UK. UMLtoCSP (UOST): A Tool For Efficient Verification of UML/OCL Class Diagrams Through Model Slicing. In *SIGSOFT FSE*, 2012; 37:1–37:4.
11. DBLP. Digital Bibliography andy Library Project. http://guifre.lsi.upc.edu/DBLP.pdf.
12. Kyriakos A, Behzad B, Geri G, Indrakshi R. UML2Alloy: A Challenging Model Transformation. In *ACM/IEEE 10th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2007)*, LNCS, 2007; 436–450.
13. Torlak E, Chang FSH, Jackson D. Finding Minimal Unsatisfiable Cores of Declarative Specifications. In *Proceedings of the 15th International Symposium on Formal Methods*, FM '08. Springer-V, 2008; 326–341.
14. Shaikh A. Formal Proof of Slicing Technique. http://www.asadshaikh.com/proof/proof.pdf.
15. Lallchandani JT, Mall R. Slicing UML Architectural Models. In *ACM / SIGSOFT SEN*, Vol. 33, 2008; 1–9.
16. Tip F. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 1995; **3**:121–189.
17. Weiser M. Program Slicing. *IEEE Trans, Software Eng* 1984:352–357.
18. Apt KR, Wallace MG. *Constraint Logic Programming Using* $\mathrm{ECL^iPS^e}$. Cambridge University Press: Cambridge, UK, 2007.
19. Cabot J, Teniente E. Determining the Structural Events That May Violate an Integrity Constraint. In *The Unified Modeling Language. Modelling Languages and Applications*, Vol. 3273, *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004; 320–334.
20. Suelflow A, Fey G, Bloem R, Drechsler R. Using Unsatisfiable Cores to Debug Multiple Design Errors. In *Proceedings of the 18th ACM great Lakes symposium on VLSI*. ACM, 2008; 77–82.
21. Kagdi HH, Maletic JI, Sutton A. Context-Free Slicing of UML Class Models. In *ICSM'05*. IEEE Computer Society, 2005; 635–638.
22. Kollmann R, Gogolla M. Metric-Based Selective Representation of UML Diagrams. In *CSMR'02*. IEEE Computer Society, 2002; 89–98.
23. Lano K, Rahimi SK. Slicing of UML Models Using Model Transformations. In *MoDELS (2)*, 2010; 228–242.
24. Altenhofen M, Hettel T, Kusterer S. OCL Support in an Industrial Environment. In *Models in Software Engineering*, Vol. 4364, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007; 169–178.
25. Uhl A, Goldschmidt T, Holzleitner M. Using an OCL Impact Analysis Algorithm for View-Based Textual Modelling. *ECEASST* 2011; **44**.
26. Goldschmidt T, Uhl A. Incremental Updates for View-Based Textual Modelling. In *ECMFA*, 2011; 172–188.

DOI: 10.1002/spe